On Improving Tunstall Codes*

Shmuel T. Klein[1] and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
[2] Dept. of Computer Science, Ashkelon Academic College, Ashkelon 78211, Israel

Abstract. Though many compression methods are based on the use of variable length codes, there has recently been a trend to search for alternatives in which the lengths of the codewords are more restricted, which can be useful for fast decoding and compressed searches. This paper explores the construction of variable-to-fixed length codes, which have been suggested long ago by Tunstall. Using new heuristics based on suffix trees, the performance of Tunstall codes can in some cases be improved by more than 40%.

1 Introduction and Background

Huffman's classical algorithm [8] designs an optimal variable length code for a given distribution of frequencies of elements to be encoded. These elements can be simple characters, in which case this is a fixed-to-variable length code, but better compression can be obtained if the text at hand can be parsed into a sequence of variable length strings, the set of which is then encoded according to the probabilities of the occurrences of its elements, yielding a variable-to-variable length encoding.

If one considers, however, also other aspects of variable length codes, not just their compression ratio, there might be incentives to revert back to fixed length codes. Decoding, for instance, is more complicated with variable length, as the end of each codeword must be determined. Variable length codewords carry usually also a processing time penalty, especially for decoding and also for other desirable features, like the possibility to search directly within the compressed text, without the need to decompress first.

This led to the development of several compromises. In a first step, the optimal binary Huffman codes were replaced by a 256-ary variant [14], in which the lengths of all the codewords are integral multiples of bytes. The loss in compression efficiency, which might be large for small alphabets, becomes tolerable and almost not significant as the alphabet size increases, in particular considering the trend set by the *Huffword* variant [13], of encoding entire words as basic elements rather than just individual characters. On the other hand, the byte-wise processing is much faster and easier to implement.

* This is an extended version of a paper that has been presented at the 15th Annual Symposium on String Processing and Information Retrieval (SPIRE'08) in Melbourne, and appeared in its Proceedings, LNCS 5280, 39–50.

When searches in the compressed text should also be supported, Huffman codes suffer from a problem of synchronization: denoting by $\mathcal{E}$ the encoding function, the compressed form $\mathcal{E}(x)$ of an element x may appear in the compressed text $\mathcal{E}(T)$, without corresponding to an actual occurrence of x in the text T, because the occurrence of $\mathcal{E}(x)$ is not necessarily aligned on codeword boundaries. A probabilistic solution to this problem has been suggested in [12]. As alternative, [14] propose to change the Huffman code by reserving the first bit of each byte as *tag*, which is used to identify the last byte of each codeword, thereby reducing the order of the Huffman tree from 256-ary to 128-ary. These *Tagged Huffman codes* have then been replaced by *End-Tagged Dense codes* (ETDC) in [3] and by (s,c)-*Dense codes* (SCDC) in [2]. An alternative code based on higher order Fibonacci numeration systems and yielding similar features is studied in [11]. The three last mentioned codes consist of fixed codewords which do not depend on the probabilities of the items to be encoded. Thus their construction is simpler than that of Huffman codes: all one has to do is to sort the items by non-increasing frequency and then assign the codewords accordingly, starting with the shortest ones.

This paper's objective is to push the idea of the compromise one step further by advocating again the use of fixed length codes. To still get reasonable compression, the elements to be encoded, rather than the codewords, will be of variable length, thus forming a variable-to-fixed length encoding. In a certain sense, this can be considered as the inverse of the basic problem solved by Huffman's algorithm. The original problem assumed that the set of elements to be encoded is given and sought for an optimal set of variable length codewords to encode those elements; the inverse problem considers the set of codewords as given, assuming furthermore that they are of fixed length, and looks for an optimal set of variable length substrings of the text which should be encoded by those codewords.

Dealing with the inverse problem can be justified by a shift in the point of view. The classical problem had as main objective to maximize the compression savings, or equivalently, using Huffman's formulation, to minimize the redundancy. The complementary approach considers as its main target a fast and easy decoding, for which a fixed length code is the best solution, but still tries to achieve good compression under these constraints. There are good reasons to view the coding problem as asymmetrical and to prefer the decoding side: in many applications, such as larger textual Information Retrieval systems, encoding is done only once while building the system, whereas decoding is repeatedly needed and directly affects the response time.

In the next section, we shall define the problem formally and also bring in previous work, in particular on Tunstall codes [18], which are variable-to-fixed length codes. In Section 3 we suggest new algorithms, compare them with other variable to fixed length codes in Section 4 and describe experimental results in Section 5.

2 Variable to fixed length encoding

Consider a text of length n characters $T = t_1 t_2 \cdots t_n$ to be encoded, where $t_i \in \Sigma$ and Σ is some general alphabet, for example ASCII. The text is to be encoded by a fixed length code in which each codeword is of length k bits, k being the only parameter of the system. The objective is to devise a dictionary $\mathcal{D}$ of different substrings of the text, such that $|\mathcal{D}| \le 2^k$ so that each of the elements of the dictionary can be assigned one of the k-bit codewords, and such that the text T can be parsed into a sequence of m dictionary elements, that is $T = c_1 c_2 \cdots c_m$, such that

$$m + \sum_{c_j \in \mathcal{D}} |c_j| \qquad (1)$$

is minimized.

The size of the encoded text will be km, which explains why one wishes to minimize the number m of elements into which the text is parsed. The reason for adding the combined size of the elements in the dictionary $\sum_{c_j \in \mathcal{D}} |c_j|$ in (1) is to avoid a bias: usually, the size of the dictionary is negligible relative to the size of the text, so that m will be dominant in (1), but without the sum, one could define one of the dictionary elements to be the entire text itself, which would yield $m = 1$.

More specifically, we are looking for an increasing sequence of integers

$$1 \le i_1 < i_2 < \cdots < i_{m-1} < i_m = n,$$

which are the indices of the last characters in the parsed elements of the text, so that $c_1 = t_1 \cdots t_{i_1}$, and for $1 < j \le m$, $c_j = t_{i_{j-1}+1} \cdots t_{i_j}$. Denote by $\ell = |\{c_j, j = 1, \ldots, m\}|$ the number of *different* strings c_j in the parsing, and by $d_1, \ldots, d_\ell$ the elements of $\mathcal{D}$ themselves. Each parsed substring c_j of the text, $1 \le j \le m$, is one of the elements d_i of the dictionary, $1 \le i \le \ell$, and the constraints are

$$\ell \le 2^k \qquad \text{and} \qquad m + \sum_{i=1}^{\ell} |d_i| \quad \text{is minimized.}$$

The number of possible partitions for fixed m is $\binom{n-1}{m-1}$, and if one sums over the possible values of m, we get $\sum_{m=1}^{n} \binom{n-1}{m-1} = 2^{n-1}$, so that an exhaustive search over the possible partitions is clearly not feasible for even moderately large texts. Choosing an optimal set of strings c_j might be intractable, since even if the strings are restricted to be the prefixes or suffixes of words in the text, the problem of finding the set is NP-complete [7], and other similar problems of devising a code have also been shown to be NP-complete in [5, 10, 6]. A natural approach is thus to suggest heuristical solutions and compare their efficiencies.

A well known variable-to-fixed length code has been suggested by Tunstall [18]. Assuming that the letters in the text appear independently from each other, the

Tunstall code[3] is iteratively built as follows: if Σ denotes the alphabet, the dictionary $\mathcal{D}$ of elements to be encoded is initialized as $\mathcal{D} \longleftarrow \Sigma$. As long as the size of $\mathcal{D}$ does not exceed 2^k, the preset size of the desired fixed length k-bit code, one then repeatedly chooses an element $d \in \mathcal{D}$ with highest probability (where the probability of a string is defined as the product of the probabilities of its constituent characters), removes it from $\mathcal{D}$ and adds all its one letter extensions instead, that is, one performs

$$\mathcal{D} \longleftarrow \mathcal{D} - \{d\} \cup \left(\bigcup_{\sigma \in \Sigma} d\sigma \right).$$

The resulting set $\mathcal{D}$ is a prefix free set, where no element is the prefix of any other, implying *unique encodability*. This property may be convenient in practical applications, but is not really necessary: even if the parsing of the text into elements of $\mathcal{D}$ can be done in more than one way, there are several possible choices of parsing heuristics for actually breaking the text into pieces, for example a greedy approach, choosing at each step the longest possible match. On the other hand, removing the constraint of unique encodability enlarges the set of potential dictionaries, which might lead to better compression.

Tunstall's procedure has been extensively analyzed [1], and the assumption of independent character appearance has been extended to sources with memory [15, 17]. Our approach is a more practical one: instead of trying to model the text and choosing the dictionary based on the expected probabilities of the strings as induced by the model, we deal directly with the substrings that actually appear in the text and which can be processed by means of the text's *suffix tree*. This can be motivated by the fact that any theoretical model of the character generation process yields only an imperfect description of natural language text. For example, a Tunstall code assuming character independence may assign a codeword to the string `eee` according to its high associated probability, even though the string might possibly not appear at all in a real text; conversely, the probabilities of positively correlated strings like `the` or `qu` will probably be underestimated. The use of a suffix tree may restrict the strings to be chosen to such that actually appear, and possibly even frequently, in the text.

On the other hand, one might object that there is a considerable effort involved in the construction and the processing of a suffix tree. Though there are algorithms that are linear in the size of the given text, once the alphabet size is considered fixed, e.g. [19], the overhead relative to the simple Tunstall construction might be larger than could be justified. But for applications where encoding is done only once, e.g., for large static IR systems as mentioned above, the additional time and space requirements might not be an issue. The details of the suggested heuristics are given in the following section.

[3] Formally, a code is a set of codewords which encode some source elements, so in our case, the code is of fixed length and consists of the 2^k possible binary k-bit strings; what has to be built by Tunstall's algorithm is not the code, but rather the *dictionary*, the set of variable length strings to be encoded.

3 A new procedure for constructing a variable-to-fixed length code

Given the text $T = t_1 \cdots t_n$, we start by forming the set $\mathcal{S}$ of all the suffixes $s_i = t_i t_{i+1} \cdots t_n\$$ of the string $T\$$, where \$ is a character not belonging to the original alphabet Σ and considered smaller than any other character to ensure lexicographical order. Each such string s_i is unique and may be used to identify the position i in the text T. The strings s_i are then stored in a *trie*, which is a labeled tree structure, as follows: every internal node of the trie has one or more children, and all edges are directed from a node to one of its children; the edges emanating from a node are labeled by different characters of Σ, ordered left to right. Every node v of the trie is *associated with* a string $s(v)$, which is obtained by concatenating, top down, the labels on the edges forming the path from the root to node v. The *suffix tree* of $T\$$ is defined as the trie for which the set of strings associated to its leaves is the set $\mathcal{S}$ of the suffixes of $T\$$. A preorder traversal of the suffix tree visits its nodes in lexicographical order of the corresponding strings.

This basic definition may yield a structure of size $\Omega(n^2)$, which can be reduced to $O(n)$ by compaction, i.e., deleting, for every node v that has only a single outgoing edge to a node w, both the node v and this edge (v, w), appending the label of the deleted edge to the right of the label of the edge e which entered v, and directing e now to point to w. Schematically, a structure

$$x \stackrel{\alpha}{\longrightarrow} v \stackrel{b}{\longrightarrow} w \qquad \text{is transformed into} \qquad x \stackrel{\alpha b}{\longrightarrow} w,$$

where $\alpha \in \Sigma^+$ denotes a string and $b \in \Sigma$ is a character. This compaction process is applied repeatedly until no nodes with single outgoing edges are left. Thus in the compacted suffix tree, also called a *suffix trie*, edges may be labeled by strings, not just characters. Suffix trees and the related suffix arrays have been used for a myriad of applications in string processing, including recently for data compression [4]. Our approach is different and will be described next.

Each node v can also be assigned a frequency $f(v)$, defined as the number of leaves of the subtree rooted at v. All the frequencies can be evaluated in a post-order traversal of the suffix tree. As mentioned, both the construction and the assignment of labels and frequencies can be done in time linear in the size n of the text.

The problem of constructing a variable to fixed length code, once the size 2^k of the set of codewords is fixed, is to choose an appropriate subset of the nodes of the suffix tree and use the corresponding labels as elements of the dictionary. The choice of the subset will be guided by the following analogy: an element that appears with probability p is ideally encoded in $-\log_2 p$ bits. This is closely approached by an arithmetic encoder, and approximated in Huffman coding, because of the additional

constraint that the length of a codeword is an integral number of bits. Looking at the relation between the probability and the corresponding codeword length, but reversing the roles of what is given (codewords of length k bits) and what we are looking for (elements to be encoded), we conclude in our case that all the strings to be chosen should have approximately probability 2^{-k}.

The frequencies $f(v)$ associated with the nodes in the suffix tree can be used to estimate the desired probabilities, but one has to be aware that several approximations are involved in the process. First, consider two strings x and y, such that a proper suffix of the first is a proper prefix of the second, in other words, there are non-empty strings x', y' and z, such that $x = x'z$ and $y = zy'$. The frequencies $f(x)$ and $f(y)$ give the number of occurrences of these strings in the text T, but not necessarily in any parsing of T into codewords. Since x and y may be overlapping, a part of the occurrences of y may not appear in the parsing. It does not even help to know the number of occurrences of the contracted superstring $x'zy'$, because the parsing is not necessarily forced to start the encoding of this string at its beginning: x' itself may have a proper prefix w, such that $x' = wx''$, which could be encoded as part of a preceding codeword, for example if $wx''zy'$ is preceded in the text by h and hw happens to be a dictionary item. This example can obviously be further extended.

Second, in order to translate the desired probabilities 2^{-k} into frequencies which can be compared to those stored in the suffix tree, one needs to multiply them by the total number of elements in the partition of T, which has been denoted in the introduction by m. However, this gives rise to a chicken and egg problem: one needs knowledge of m to evaluate the frequencies, with the help of which an appropriate subset of nodes can be selected; the corresponding strings then form the dictionary and induce a partition of the text T into m' occurrences of the dictionary terms. There is no guarantee that $m = m'$.

We still shall base our heuristic on the frequencies $f(v)$, but do not claim that these values reflect the actual number of occurrences. They can, nevertheless, serve as some rough estimate if one assumes that the overlaps mentioned above, which will bias the counts, are spread evenly over all the processed strings. To describe the heuristic, we need some definitions.

Definition 1: Given a compacted suffix tree, we define a *cut* C of the tree as a line crossing the path from the root to each of the leaves at exactly one edge.

Definition 2: The *lower border* of a cut, $LB(C)$, is the set of nodes of the suffix tree just below the cut, where we refer to the convention of drawing (suffix) trees with the root on the top and with edges leading from a parent to a child node pointing downwards.

Definition 3: The *upper border* of a cut, $UB(C)$, is the set of nodes of the suffix tree which are parent nodes of at least one node of the lower border.

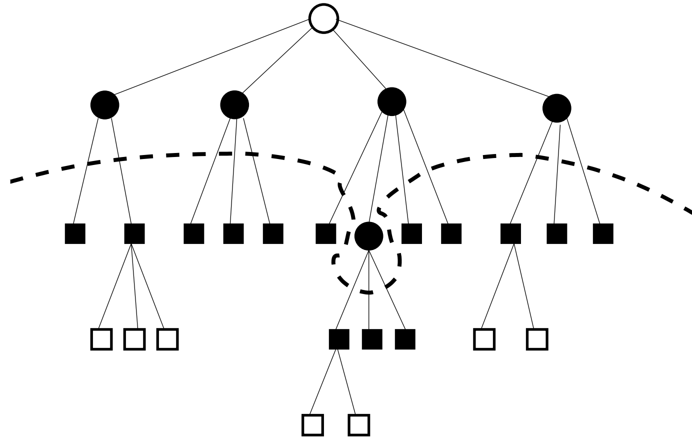

FIGURE 1: Schematic representation of a cut in a suffix tree

Figure 1 is a schematic representation of a small suffix tree visualizing these definitions. The cut is the broken line traversing the full breadth of the tree. Nodes above the cut are drawn as circles and those below the cut as squares. The nodes of the borders are filled with color (black squares for the lower border and black circles for the upper one), and those not belonging to any of the borders are only outlined.

The following properties will be useful below.

Theorem: Given a compacted suffix tree with n leaves, and any cut C of the tree, we have

$$\sum_{v \in LB(C)} f(v) = n,$$

that is, the sum of the frequencies of the nodes of the lower border of all possible cuts is constant, and equal to the size of the underlying text n.

Proof: Each node in the lower border $LB(C)$ is either a leaf node, or it is the root r of a subtree R for which $f(r)$ is the number of leaves in the tree R. Since all the leaves are counted exactly once in the sum, the result follows. ∎

Observation: Given a compacted suffix tree with n leaves, and any cut C of the tree, the strings associated with the nodes of the lower border $LB(C)$ form a dictionary $\mathcal{D}$ which ensures unique encodability.

Proof: For the fact that at most one encoding is possible, it suffices to show that the strings form a prefix set. Assume on the contrary that there are two strings v and w in $\mathcal{D}$ such that v is a prefix of w, and denote the corresponding nodes in the tree by n_v and n_w, respectively. Then n_v is an ancestor node of n_w in the tree, and since both nodes belong to $LB(C)$, the cut C crosses the path from the root to n_w twice: once at the edge entering n_w and once at the edge entering n_v, in contradiction with the definition of a cut. Thus no string of $\mathcal{D}$ is the prefix of any other.

At least one encoding is always possible due to the *completeness* of the set, in the sense that a cut has been defined as a line crossing every path in tree. As a constructive proof, suppose that the prefix of length $i-1$ of T has already been uniquely encoded, we show that exactly one codeword can be parsed starting at the beginning of the remaining suffix $t_i t_{i+1} \cdots$. Consider a pointer pointing to the root of the suffix tree, and use the characters $t_i t_{i+1} \cdots$ to be processed as guides to traverse the tree. For each character x read, follow the edge emanating from the current node and labeled by x. Such an edge must exist, because the tree reflects all the substrings that appear in the text. This procedure of stepping deeper into the tree at each iteration must ultimately cross the cut C, and the string associated with the first node encountered after crossing the cut, is the next element in the parsing. ∎

Note that the strings associated with the *upper* border of a cut do not always form a prefix set, as can be seen in the example in Figure 1.

From the observation we learn that it might be a good idea to define the dictionary as the lower border of some cut, so we should look for cuts C for which $|LB(C)| = 2^k$. Our first heuristic, which we call STT for Suffix-Trie-Tunstall, extends the Tunstall procedure, but working top-down on the compacted suffix tree with actual frequencies instead of an artificial tree with estimated probabilities.

<u>STT</u>: Top-down construction of $\mathcal{D}$

$\mathcal{D} \longleftarrow$ nodes on level 1 of suffix trie
while $|\mathcal{D}| < 2^k$
  $v \longleftarrow$ element of $\mathcal{D}$ with maximal $f(v)$
  $w_1, \ldots, w_r \longleftarrow$ children of v
  $\mathcal{D} \longleftarrow \mathcal{D} - \{v\} \cup \{w_1, \ldots, w_r\}$
end while
if $|\mathcal{D}| > 2^k$ undo last assignment

As alternative, the second heuristic, which we call DynC for Dynamic Cut, also works on the suffix trie, but traverses it left to right and constructs the lower border of the desired cut according to the local frequencies. One of the problems mentioned earlier was that one cannot estimate the frequencies without knowing their total sum. But because of the above Theorem, we know that if we restrict ourselves to choose the elements of the lower border of a cut, the sum of all frequencies will remain constant. We can therefore look for nodes v in the tree for which $f(v)/n \simeq 2^{-k}$, that is $f(v) \simeq n2^{-k}$.

Ideally, there should be 2^k such elements, but in practice, there is a great variability in the frequencies. We therefore suggest to build the dictionary incrementally in a left to right scan of the tree, adapting the target value of the desired frequency

for the current element dynamically, according to the cumulative frequencies of the elements that are already in the dictionary. More formally:

DynC: Left-to-right construction of $\mathcal{D}$

$\mathcal{D} \longleftarrow \emptyset$
target $\longleftarrow n2^{-k}$
cumul $\longleftarrow 0$
scan the suffix trie in DFS order while $|\mathcal{D}| < 2^k$
  $v \longleftarrow$ next node in scan order for which $f(v) \le$ target
  $\mathcal{D} \longleftarrow \mathcal{D} \cup \{v\}$
  cumul $\longleftarrow$ cumul $+f(v)$
  target $\longleftarrow \dfrac{n - \text{cumul}}{2^k - |\mathcal{D}|}$
end while

The updated target value is obtained by dividing the expected sum of the frequencies of the remaining elements to be added by their number. This allows the procedure to set the target higher than initially, in case some elements have been chosen with very low occurrence frequency.

The strict compliance with the constraints imposed by deciding that $\mathcal{D}$ should be a complete prefix set turned out to be too restrictive. In many cases, a node with quite high frequency could have several children with very low occurrence rate, and including the strings associated with these children nodes in $\mathcal{D}$ would eventually clog the dictionary with many strings that are practically useless for compression. To avoid the bias caused by the low values, a lower bound B has been imposed on $f(v)$ for the string $s(v)$ to be considered as a candidate to be included in $\mathcal{D}$. As a result, the dictionary was not complete any more, so to ensure that the text can be parsed in at least one way, $\mathcal{D}$ was initialized with the set of single characters. This, in turn, implied the loss of the prefix property, so the parsing with the help of the suffix tree needed to be supplemented with some heuristic, for example a greedy one, trying at each step to parse the longest possible dictionary element.

Decoding of the fixed length code is of course extremely simple. All one has to do is to store the strings of $\mathcal{D}$ consecutively as a string S, and refer to each element by its offset and length in the string S. These (*off*,*len*) pairs are stored in the dictionary table DT, which is accessed by 2-byte indices (in the case $k = 16$) forming the compressed text. Formally:

Decoding of STT or DynC encoded text

while ($i \longleftarrow$ read next 2 bytes) succeeds
  $(\mathit{off}, \mathit{len}) \longleftarrow DT[i]$
  output $S[\mathit{off} \cdot\cdot \mathit{off}{+}\mathit{len}{-}1]$

For $k = 12$, in order to keep byte alignment, one could process the compressed text by blocks of 3 bytes, each of which decodes to two dictionary elements.

4 Comparison with other variable-to-fixed length codes

We now turn to a comparison with other variable-to-fixed length schemes, in particular those based on the various Lempel-Ziv techniques, though these are usually implemented with variable length codings. But the encodings can easily be transformed to be of fixed length, yielding a tradeoff between compression efficiency and decoding simplicity.

One of the Lempel-Ziv algorithms [24], known as LZ78, and its popular variant LZW [21] are based on parsing the text into phrases belonging to a dictionary, which is dynamically built during the parsing process itself. The output of LZW consists of a sequence of pointers, and the size of the encoding of the pointers is usually adapted to the growing dictionary: starting with a dictionary of 512 entries, one uses 9-bit pointers to refer to its elements, until the dictionary fills up. At that stage, the number of potential entries is doubled and the length of the pointers is increased by 1. This procedure is repeated until the size of the dictionary reaches some predetermined limit, say 64K with 16-bit pointers. In principle, the dictionary could be extended even further, but erasing it and starting over from scratch has the advantage of allowing improved adaptation to dynamically changing texts, while only marginally hurting the compression efficiency on many typical texts. A fixed length encoding variant of LZW could thus fix the size $|\mathcal{D}|$ of the dictionary in advance and use throughout $\log |\mathcal{D}|$ bits for each of the pointers.

The other LZ algorithm [23], known as LZ77, produces an output consisting of a strictly alternating sequence of single characters and pointers, but no external dictionary is involved and the pointers, consisting of (offset, length) pairs, refer to the previously scanned text itself. The variant in [16] suggests to replace strict alternation by a set of flag-bits indicating whether the next element is a single character or an (offset, length) pair. A simple implementation, like in [22] uses 8 bits to encode a character, 12 bits for the offset and 4 bits for the length. Excluding the flag-bits, we thus get codewords of lengths 8 or 16. To get a fixed length encoding, the length of the shortest character sequence to be copied is set to 3 (rather than 2 in the original algorithm); thus when looking for the longest previously occurring sequence P which

matches the sequence of characters C starting at the current point in the text, if the length of P is less than 3, we encode the first 2 characters of C. In the original algorithm, if the length of P was less than 2, we encoded only the first character of C. The resulting encoding therefore uses only 16 bits elements, regardless of if they represent character pairs or (offset, length) pairs. To efficiently deal also with the flag bits, one can use the same technique as in [22], aggregating them into blocks of 16 elements each.

One of the challenges of LZ77 is the way to locate the longest matching previously occurring substring. Various techniques have been suggested, approximating the requested longest match by means of trees or hashing. We use here an adaptation of the fast method suggested in [22], replacing the hashing of character pairs by a lookup in a table of size 2^{16} and extending a previous occurrence as much as possible. Note, however, that the fixed length variant of LZ77 is no real competitor to Tunstall or the suggested heuristics, since there is no single fixed dictionary that can be used throughout. It is thus not possible to decode only selected short fragments.

There are several criteria by which different coding techniques could be compared. Those mentioned in [2] are compression ratio, encoding and decoding speed, and the facility to perform compressed matching, that is, searching for sub-strings directly in the compressed file. For our current study, we shall restrict attention only to compression efficiency and decoding time. Encoding is supposed to be done offline, as mentioned earlier, and in applications for which the encoding time is also important, one can hardly justify a heuristic using a suffix tree. As to compressed matching, all variable to fixed length encodings would perform equally badly: the problem is that a pattern to be searched for is not necessarily restricted to be one of the elements of the dictionary at hand, but could appear as a substring of the concatenation of several elements, and the number of relevant concatenations might be exponential in the length of the pattern.

5 Experimental results

To empirically test the suggested heuristics, we chose the following input files of different sizes and languages: the Bible (King James version) in English, the French version of the European Union's JOC corpus, a collection of pairs of questions and answers on various topics used in the ARCADE evaluation project [20], and the concatenated text of all the XML files of the INEX database [9]. To get also different alphabet sizes, the Bible text was stripped of all punctuation signs, leaving only blank, upper and lower case letters[4], whereas the other texts have not been altered[5].

Table 1 compares the compression efficiency of the suggested heuristics with that of the Tunstall codes, for both $k = 12$, corresponding to a small dictionary of 4096

[4] Uppercase X did not appear in the text, hence a total of only 52 characters.
[5] the French text included many accented letters, which explains the large alphabet.

File	Bits	Tunstall expected	Tunstall actual	STT	DynC B	DynC compression
English 2.96 MB 52 chars	12	0.617	0.614	**0.475**	160	0.488
					190	0.481
					220	0.497
	16	0.589	0.587	**0.334**	8	0.411
					11	0.395
					14	0.428
French 7.26 MB 131 chars	12	0.744	0.751	0.568	300	0.526
					450	**0.496**
					600	0.547
	16	0.691	0.689	**0.384**	15	0.416
					22	0.401
					30	0.410
XML 494.7 MB 94 chars	12	0.745	0.754	0.691	20000	0.607
					30000	**0.594**
					40000	0.646
	16	0.710	0.709	**0.481**	800	0.505
					900	0.493
					1000	0.495

TABLE 1: *Comparison of compression performance between Tunstall,* STT *and* DynC

entries, and $k = 16$, for a larger dictionary with 65536 elements. The first column lists also relevant statistics, the size of the files and the size of the alphabets, and the second column gives the parameter k. All compression figures are given as the ratio of the size of the compressed file to the full size before compression. The column headed *expected* gives the expected compression of the Tunstall code: let $A = \sum_{v \in \mathcal{L}} p_v \ell_v$ be the average length of a Tunstall dictionary string, where $\mathcal{L}$ is the set of leaves of the Tunstall tree, p_v is the probability corresponding to leaf v, and ℓ_v is its depth in the tree. Then n/A is the expected number of codewords used for the encoding of the text, so the expected size of the compressed text is $kn/8A$, and the compression ratio is $k/8A$. As noted above, the Tunstall dictionary contains many strings that are not really used. The column headed *actual* is the result of actually parsing the text with the given dictionary, giving quite similar results.

A significant improvement can be observed for the use of STT versus that of Tunstall, and for the larger dictionaries, with $k = 16$, STT was even able to cut the Tunstall encoded files to about 55–68% of their size. The results for DynC are given for several bounds. Interestingly, on all our examples, compression first improves with increasing bound, reaches some optimum, and then drops again. This can be explained by the fact that increasing the bound leads to longer strings in the dictionary, but increasing it too much will imply the loss of too many useful shorter strings. Table 1 shows three examples of different bounds B for each file and each

k. One can see that DynC reduces the file by additional 20–40% relative to Tunstall, and on the smaller dictionaries, with $k = 12$, may improve also over STT for certain values of the bound B. The best values for each category are boldfaced.

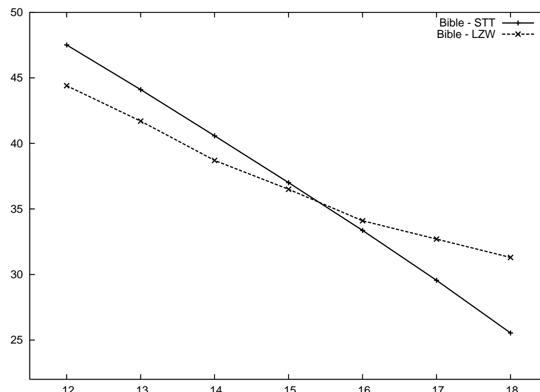

FIGURE 2: Comparing STT with fixed LZW compression as function of the number of encoding bits

The graph in Figure 2 compares the performance of STT with that of a fixed length LZW variant, as described above, for encodings of lengths 12 to 18 bits. To enable similar decoding conditions, the dictionary of LZW has been chosen to be fixed, so once it filled up in the initial stage, it did not change any more. The plotted values are the relative sizes, in percent, of the compressed files relative to the uncompressed ones, and correspond to the English text used in Table 1. We see that while both curves are clearly decreasing, the fixed length LZW is preferable up to 15 bits encodings, but for larger dictionaries, STT is clearly better.

	Tunstall	Huffman	LZ77	LZW	STT	DynC	SCDC	Fib3
English	100	89.9	112.7	58.1	56.7	67.1	43.1	39.2
French	100	83.9	105.9	50.2	55.6	58.2	44.6	38.8
XML	100	88.5	89.7	68.2	67.8	69.5	57.8	51.7

TABLE 2: *Comparison of compression performance of* STT *and* DynC *with other methods*

A comparison of STT and DynC with other methods can be found in Table 2. The sizes are given as a percentage of the size of the file compressed by Tunstall, corresponding to 100%, with a dictionary of 2^{16} entries. Regular Huffman coding, based on the individual characters, is only 10–12% better than Tunstall. The values for LZW and LZ77 correspond also to 16-bit codewords, as explained in Section 4 above. Note that the compression figures by LZ77 are actually worse than those of Tunstall for the French and English files, and are brought here only for reference, since the method is inherently adaptive and thus cannot compete with the others,

which are all based on static dictionaries. The values for DynC correspond to the parameter B that gave the best results in Table 1. On the given examples, DynC is inferior to STT and even to LZW, but we saw in Table 1 that this is not always the case.

Still better compression can be obtained if one does not insist on fixed length codes, as for SCDC [2], where the values are given for the best (s, c) pair, or Fib3 [11]. These last two methods are only mentioned to give an idea of the compressibility of the test files and cannot directly be compared with the other methods, because SCDC and Fib3 are based on encoding the different words of the text, rather than the characters. We here limited the number of encoded elements also to 2^{16}, which is more than the number of words in the Bible. For the French and XML texts, the excess words were encoded letter by letter, so that the total dictionary consisted of 2^{16} elements.

	Huffman	$k = 12$			$k = 16$		
		Tunstall	STT	DynC	Tunstall	STT	DynC
English	0.35	0.14	0.12	0.12	0.16	0.14	0.12
French	0.93	0.38	0.33	0.31	0.32	0.24	0.25
XML	68.87	26.56	25.36	24.32	21.52	18.63	18.55

TABLE 3: *Comparison of decoding time*

Table 3 is a comparison of timing results, measuring the decoding time of the entire file. Values are given in seconds. We see that the decoding time is roughly proportional to the size of the file for the fixed length codes, and thus slightly better for STT and DynC than for Tunstall. On the other hand, the decoding of the variable length Huffman codes is more involved and takes on our tests about 2.5 times longer, even though the files are shorter than for the corresponding fixed length encodings.

We conclude that if one has good reasons to trade compression efficiency for the simplicity of fixed length codes, the suggested heuristics may be a worthwhile alternative to the classical Tunstall codes.

References

[1] ABRAHAMS J., Code and parse trees for lossless source encoding, *Comm. in Information and Systems* **1**(2) (2001) 113–146.

[2] BRISABOA N.R., FARIÑA A., NAVARRO G., ESTELLER M.F., (*s*,*c*)-dense coding: an optimized compression code for natural language text databases, *Proc. Symposium on String Processing and Information Retrieval SPIRE'03*, *LNCS* **2857**, Springer Verlag (2003) 122–136.

[3] BRISABOA N.R., IGLESIAS E.L., NAVARRO G., PARAMÁ J.R., An efficient compression code for text databases, *Proc. European Conference on Informa-*

tion Retrieval ECIR'03, Pisa, Italy, *LNCS* **2633**, Springer Verlag (2003) 468–481.

[4] Crochemore M., Ilie L., Smyth W.F.,, A Simple Algorithm for Computing the Lempel Ziv Factorization, *Proc. Data Compression Conference DCC–2008*, Snowbird, Utah (2008) 482–488.

[5] Fraenkel A.S., Klein S.T., Complexity Aspects of Guessing Prefix Codes, *Algorithmica* **12** (1994) 409–419.

[6] Chrobak M., Kolman P., Sgall J., The greedy algorithm for the minimum common string partition problem *ACM Transactions on Algorithms* **1**(2) (2005) 350–366.

[7] Fraenkel A.S., Mor M., Perl Y., Is text compression by prefixes and suffixes practical? *Acta Informatica* **20** (1983) 371–389.

[8] Huffman D., A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40** (1952) 1098–1101.

[9] Kazai G., Gövert N., Lalmas M., Fuhr N., The INEX Evaluation Initiative, in *Intelligent Search on XML data*, LNCS **2818** (2003) 279–293.

[10] Klein S.T., Improving static compression schemes by alphabet extension, *Proc. 11th Symp. on Combinatorial Pattern Matching*, Montreal, Canada, *Lecture Notes in Computer Science* **1848**, Springer Verlag, Berlin (2000) 210–221.

[11] Klein S.T., Kopel Ben-Nissan M., On the Usefulness of Fibonacci Compression Codes, to appear in *The Computer Journal* **52** (2009) doi:10.1093/comjnl/bxp046.

[12] Klein S.T., Shapira D., Pattern matching in Huffman encoded texts, *Information Processing & Management* **41**(4) (2005) 829–841.

[13] Moffat A., Word-based text compression *Software – Practice & Experience* **19** (1989) 185–198.

[14] de Moura E.S., Navarro G., Ziviani N., Baeza-Yates R., Fast and flexible word searching on compressed text, *ACM Trans. on Information Systems* **18** (2000) 113–139.

[15] Savari S.A., Gallager R.G., Generalized Tunstall codes for sources with memory, *IEEE Trans. Info. Theory* **IT–43** (1997) 658–668.

[16] Storer J.A., Szymanski T.G., Data Compression Via Textual Substitution, *Journal of the ACM*, **29**(4) (1982) 928–951.

[17] Tjalkens T.J., Willems F.M.J., Variable to fixed length codes for Markov sources, *IEEE Trans. Info. Theory* **IT–33** (1987) 246–257.

[18] Tunstall B.P., Synthesis of noiseless compression codes, PhD dissertation, Georgia Institute of Technology, Atlanta, GA (1967).

[19] Ukkonen E., On-line construction of suffix trees, *Algorithmica* **14**(3) (1995) 249–260.

[20] Véronis, J., Langlais, P., Evaluation of parallel text alignment systems: The arcade project, in *Parallel Text Processing*, J. Véronis, ed., Kluwer Academic Publishers, Dordrecht (2000) 369–388.

[21] WELCH T.A., A technique for high performance data compression, *IEEE Computer* **17** (1984) 8–19.

[22] WILLIAMS R.N., An extremely fast Ziv-Lempel data compression algorithm, *Proc. Data Compression Conference DCC–91*, Snowbird, Utah (1991) 362–371.

[23] ZIV J. AND LEMPEL A., A Universal Algorithm for sequential data compression, *IEEE Trans. on Information Theory* **23** (1977) 337–343.

[24] ZIV J. AND LEMPEL A., Compression of individual sequence via variable rate coding, *IEEE Trans. on Information Theory* **24** (1978) 530–536.