A New Approach to Alphabet Extension for Improving Static Compression Schemes

Shmuel T. Klein

Department of Computer Science
Bar Ilan University, Ramat-Gan 52900, Israel
Tel: (972–3) 531 8865 Fax: (972–3) 736 0498
tomi@cs.biu.ac.il

Abstract: The performance of data compression on a large static text may be improved if certain variable-length strings are included in the character set for which a code is generated. A new method for extending the alphabet is presented, based on a reduction to a graph-theoretic problem. A related optimization problem is shown to be NP-complete, a fast heuristic is suggested, and experimental results are presented.

1 Introduction and Background

Compression methods may be classified according to various criteria, such as compression efficiency, speed, necessary space in RAM, etc. (see [32] or [6] for an overview). The present work focuses on a static compression scheme, for which we assume that the time restrictions on the encoding and decoding processes are not symmetrical: we shall put no constraint on compression time, but require very fast decompression. The corresponding scenario is that of a large full-text information retrieval (IR) system, for which compression is usually performed only once or periodically (but rarely), but which may be accessed frequently, so that (possibly many) small parts of it need to be decompressed while the user is waiting for responses to a query. The text of the IR system may be static, like for the Encyclopædia Britannica, the *Trésor de la Langue Française* [11], the *Responsa Retrieval Project* (RRP) [16, 14], etc., or it might be dynamically increasing over time, like collections of news wire messages, newspapers, and ultimately, the whole World Wide Web. The RRP mentioned above, one of the oldest IR systems, was headed by Prof. Yaacov Choueka for many years and was a major incentive for the present, as well as many related works.

Statistical compression methods, like Huffman or arithmetic coding, assume that the text at hand is a sequence of data items, that will be encoded according to their occurrence frequencies. These data items are generally single characters, but sometimes more sophisticated models are used for which a character pair, a word, or more generally, certain character strings, are considered to be an item. *Dictionary* compression methods work by replacing the occurrence of certain strings in the text by (shorter) pointers to some dictionary, which again may be either static, fixed in advance (e.g., the most frequent words in English), or dynamically adapted to the given text, like the various Lempel-Ziv methods and their variants. The statistical methods working only on single characters are often inferior to the dictionary methods from the compression point of view. Nevertheless, they may be the preferred choice in IR applications, as they do not require a sequential scan of the compressed file.

To improve the performance of the statistical methods, much larger alphabets could be defined, as in the *Huffword* variant [17], in which every word, rather than every character, is considered as an atomic element to be encoded. Another line of investigation suggested to trade a part of the compression efficiency against better processing abilities, including faster decoding and the possibility to search directly in the compressed text [28, 12, 24]. Using large alphabets, the reduction in compression is only a few percent, but decoding and searches are much faster.

The focus in this paper is again the compression ratio of statistical methods, and we shall follow the approach in [9] and [10], and construct a *meta-alphabet*, which extends the standard alphabet by including also frequent variable-length strings. The idea is that even if we assume a quite involved model for the text generation process, there will always be strings deviating from this model. One may thus improve the accuracy of the model by including such strings as indivisible items, called below *meta-characters,* into the extended alphabet.

Obviously, the larger the meta-alphabet, the better compression we may expect. The problem then is that of an optimal exploitation of a given amount of RAM, which puts a bound on the size of the dictionary. We shall assume that this resource is limited, so that decompression at least should be possible even on very weak machines. In addition, there is a problem with the selection of the set of meta-characters, because the potential strings are overlapping. A similar problem has been shown to be NP-complete under certain constraints [18]. Several efficient greedy heuristics have been suggested in [3, 4]. They are based on iteratively choosing the most promising string, and substituting all its non-overlapping occurrences, except one, by a new meta-character. Other works addressing the problem of compressing using a fixed alphabet can be found in [26, 13, 8].

Our approach concentrates on the meta-alphabet itself, rather than on the individual occurrences of its elements in the text. This enables a clean separation of the modeling (definition of the meta-alphabet) from the actual encoding (choice of encoding and parsing methods), as advocated by [7]. Such a separation allows the evaluation of the contribution of each part independently from the other, by fixing the model and varying the encoding, or alternatively, by using a given encoding scheme and applying it to various models. We strive in this work to optimize the model or at least to make sound decisions in successive approximations of such an optimum.

The paper is organized as follows: in the next section, we present a novel approach to alphabet extension and show that a related optimization problem is NP-complete. Section 3 deals with implementation issues and refinements and suggests a fast heuristic, and Section 4 mentions some possible extensions. Finally, we bring some experimental results in Section 5.

2 Definition of Meta-alphabet

The criterion for including a string of characters as a new meta-character into the meta-alphabet is the expected savings we would incur by its inclusion. The exact value is hard to evaluate, since the savings depend ultimately on the encoding method, and for Huffman codes, say, the length of each codeword may depend on all the others. Assume for simplicity in a first stage, that we shall use a fixed length code to refer to any element of the meta-alphabet $\mathcal{A}$, which contains both single characters and the meta-characters. If $|\mathcal{A}| = D$, any element can be encoded by $\lceil \log_2(D) \rceil$ bits.

Our approach starts by a reduction to a graph-theoretic problem in the following way. There will be a vertex for every possible character string in the text, and vertices are connected by an edge if the corresponding strings are overlapping. Both vertices and edges are assigned weights. The weight $w(x)$ of a vertex x will be the *savings*, measured in number of characters, obtained by including x in $\mathcal{A}$. The weight $w(x, y)$ of an edge (x, y) will be the *loss* of such savings due to the overlap between x and y. We are thus interested in a subset V' of the vertices, not larger than some predetermined constant K, which maximizes the overall savings

$$\sum_{x \in V'} w(x) \; - \sum_{x,y \in V'} w(x, y). \tag{1}$$

Formally, we are given a text $T = t_1 t_2 \cdots t_n$, where each t_i belongs to a finite alphabet Σ; define a directed graph $G = (V, E)$, where V is the set of all the substrings of T, i.e., the strings $t_i \cdots t_j$ for $1 \leq i \leq j \leq n$, and there is a directed edge from $x = x_1 \cdots x_k$ to $y = y_1 \cdots y_\ell$ if some suffix of x is a prefix of y, i.e., there is a $t \geq 1$ such that $x_{k-t+j} = y_j$ for all $1 \leq j \leq t$. For example, there will be a directed edge from the string `element` to `mention`, corresponding to $t = 4$. The weight of a vertex x is defined as

$$w(x) = \mathit{freq}(x)(|x| - 1),$$

where $\mathit{freq}(x)$ is the number of occurrences of the string x in T, $|a|$ denotes the length (number of characters) of a string a, and the -1 accounts for the fact that if the string x is selected as a meta-character, then for each occurrence of x, $|x|$ characters are saved, but one meta-character will be used, so we save only $|x|-1$ characters (recall that we assume a fixed-length code, so that the characters and meta-characters are encoded by the same number of bits).

For strings x and y like above, define the super-string, denoted $\overline{xy}$, as the (shortest) concatenation of x with y, but without repeating the overlapping part, i.e., $\overline{xy} = x_1 \cdots x_k y_{t+1} \cdots y_\ell$, where t has been chosen as largest among all possible t's. For example, if x is `element` and y is `mention`, then $\overline{xy}$ is `elemention`. The weight of the directed edge from x to y is defined as

$$w(x, y) = \mathit{freq}(\overline{xy})(|y| - 1).$$

The reason for this definition is that if the text will be parsed by a greedy method, it may happen that $\mathit{freq}(\overline{xy})$ of the $\mathit{freq}(y)$ occurrences of y will not be detected, because for these occurrences, the first few characters of y will be parsed as part of x.

In fact, assuming that the parsing of $\overline{xy}$ will always start with x in a greedy method is an approximation. For it may happen that certain occurrences of x will stay undetected because of another preceding string z, that has a non-empty suffix overlapping with a prefix of x. To continue the example, suppose z is `steel`, then $\overline{z\overline{xy}}$ is `steelemention`, which could be parsed as z`e`y. Moreover, when the overlapping part itself has a prefix which is also a suffix, then choosing the *shortest* concatenation in the definition of $\overline{xy}$ does not always correspond to the only possible sequence of characters in the text. For example, if x is `managing` and y is `ginger`, $\overline{xy}$ would be `managinger`, but the text could include a string like `managinginger`. We shall however ignore such cases and keep the above definition of $w(x, y)$.

A first simplification results from the fact that we seek the subgraph induced by the set of vertices V', so that either both directed edges (x, y) and (y, x) will be included, if they both exist, or none of them. We can therefore consider an equivalent undirected graph, defining the label on the (undirected) edge (x, y) as

$$w(x, y) \ = \ \mathit{freq}(\overline{xy})(|y| - 1) \ + \ \mathit{freq}(\overline{yx})(|x| - 1).$$

For a text of n characters, the resulting graph has $\Theta(n^2)$ vertices, and may thus have $\Theta(n^4)$ edges, which is prohibitive, even for small n. We shall thus try to exclude a priori strings that will probably not be chosen as meta-characters. The excluded strings are:

1. a string of length 1 (they are included anyway in $\mathcal{A}$);
2. a string that appears only once in the text (no savings can result from these).

For example, consider the text

`the-car-on-the-left-hit-the-car-i-left-on-the-road`.

Using the above criteria reduces the set of potential strings to: {`the-car-`, `-on-the-`, `-left-`}, and all their substrings of length > 1. If this seems still too large, we might wish to exclude also

3. a string x that always appears as a substring of the same string y.

This would then purge the proper substrings from the above set, except the string `the-`, which appears as substring of *different* strings. The rationale for this last criterion is that it is generally preferable to include the longer string y into the extended alphabet. This is, however, not always true, because a longer string has potentially more overlaps with other strings, which might result in an overall loss, as will be shown below.

Figure 1 depicts the graph of the above example, but to which we have added `on-the-` as fifth string to the alphabet (in spite of it appearing always as substring of `-on-the-`) because it will yield an example showing a deficiency of criterion 3. Edges with weight 0 have been omitted, so there is no edge

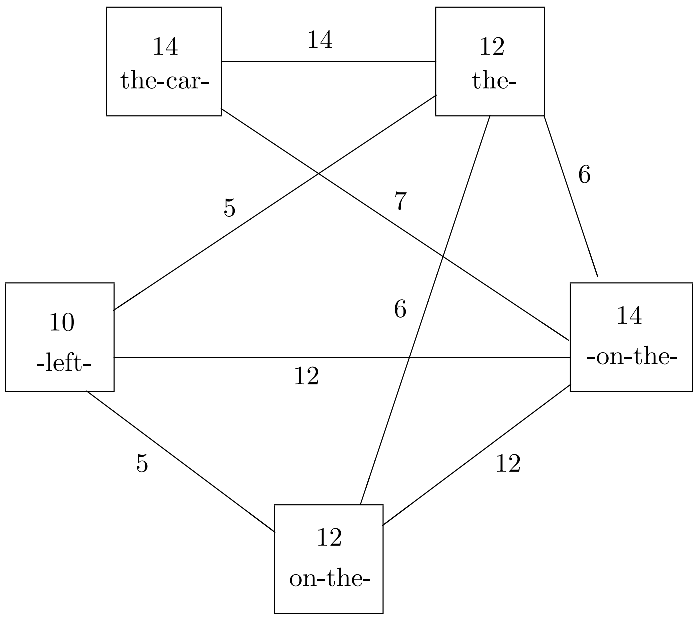

FIGURE 1: *Graph for text*
`the-car-on-the-left-hit-the-car-i-left-on-the-road`

between `-left-` and `the-car-`, or between `on-the-` and `the-car-`, because the super-strings `the-car-left-` and `on-the-car-` do not appear; similarly, there is no self-loop on the vertex x =`-left-`: even though the super-string $\overline{xx}$ =`-left-left-` is defined, its frequency is zero. If we are looking for a set of 3 meta-characters, the best of the $\binom{5}{3} = 10$ triplets is `the-car-`, `-left-`, `on-the-`, yielding savings of $14 + 10 + 12 - 5 = 31$ characters. Indeed, replacing all occurrences of the meta-characters in the text reduces the number of elements in the parsing from 50 to 19, of which 14 are single characters and 5 are meta-characters. However, one can see here that the criterion of excluding a string x if it always appears as substring of y, does not assure optimality: if we would choose the string `on-the` instead of `on-the-` as fifth vertex, the text could be parsed as 16 instead of 19 elements.

There is, however, still a problem with the complexity of the algorithm. A similar problem has been shown to be NP-complete in [30], but we bring here a direct proof:

Theorem 1. *The problem of finding a subgraph maximizing (1) is NP-complete.*

Proof: One actually has to deal with the corresponding decision problem, which we denote SDP (Substring Dictionary Problem). Given a graph $G = (V, E)$, with weights on both vertices and edges, and 3 non-negative constants K_1, K_2 and

K_3, is there a subset of vertices $V' \subseteq V$, such that $|V'| \leq K_1$, $\sum_{x \in V'} w(x) \geq K_2$ and $\sum_{x,y \in V'} w(x,y) \leq K_3$?

Once a guessing module finds the set V', the other conditions are easily checked in polynomial time, so SDP $\in$ NP. To show that SDP is also NP-hard, the reduction is from Independent Set (IS) [19], defined by: given a graph $G_1 = (V_1, E_1)$ and an integer L, does G_1 contain an independent set of size at least L, that is, does there exist a subset $V_1' \subseteq V_1$, such that $|V_1'| \geq L$, and such that if x and y are both in V_1', then $(x,y) \notin E_1$?

Given a general instance of IS, define the following instance of SDP: let $G = G_1$, define $w(x) = 1$ for all vertices $x \in V$, $w(x,y) = 1$ for all edges $(x,y) \in E$, $K_1 = |V|$, $K_2 = L$ and $K_3 = 0$. Suppose that there is an independent set V_1' of size at least L in G_1. We claim that the same set also satisfies the constraints for SDP. Indeed, $|V_1'| = L \leq K_1$, $\sum_{x \in V_1'} w(x) = |V_1'| = L \geq K_2$, and since in an independent set, there are no edges between the vertices, $\sum_{x,y \in V_1'} w(x,y) = 0 \leq K_3$.

Conversely, suppose there is a set $V' \subseteq V$ which fulfills the conditions of SDP in the graph G. Then because the weight of each vertex is 1, it follows from the second condition that there are at least $K_2 = L$ vertices in V'. The choice of K_3 as 0 in the third condition, together with the fact that the weight of each edge is 1, implies that no two vertices of V' may be connected by an edge, that is, V' is an independent set. ∎

Two problems have therefore to be dealt with: first, we need a fast procedure for the construction of the graph, and second we seek a reasonable heuristic, running fast enough to yield a practical algorithm, and still making better choices than discarding any strings that overlap with one previously chosen.

3 Implementation Details

3.1 Graph Construction

Since we are looking for a special set of substrings of the given input text T, a *position-tree* or *suffix-tree* [5, 31] may be the data structure of choice for our application, as it is in similar algorithms like, e.g., in [4]. The strings occurring at least twice correspond to the internal nodes of the tree. As to condition 3. above, if a string x occurs always as a prefix of y, the node corresponding to x has only one child in the position tree, and will therefore not appear as a node in the *compacted* form of the tree. Seeking the longest re-occurring strings, these correspond, for each branch of the tree, to the lowest level of the internal nodes, i.e., the parent nodes of the leaves. The other internal nodes of the compacted tree correspond to strings that are prefixes of at least two different strings. However, the set of strings corresponding to the internal nodes of the compacted position tree might include more strings than those defined by conditions 1.–3., e.g., a string x that always appears as *suffix* of some other string y. Such strings could be purged by using an auxiliary position tree, built for the reversed input string, but empirical tests have shown that this may not be worth the effort.

The advantage of defining the set of vertices as corresponding to the internal nodes of the position tree is that we get n, the length of the text, as immediate bound for the number of vertices. Indeed, the branching factor of every internal node in the compacted position tree is at least 2 (and for many nodes much larger than that), so in the worst case, we have a full binary tree with n leaves, and therefore $n-1$ internal nodes. Figure 2 shows a part of the compacted position tree for our example, where the black vertices correspond to the four strings left after applying conditions 1.–3. above.

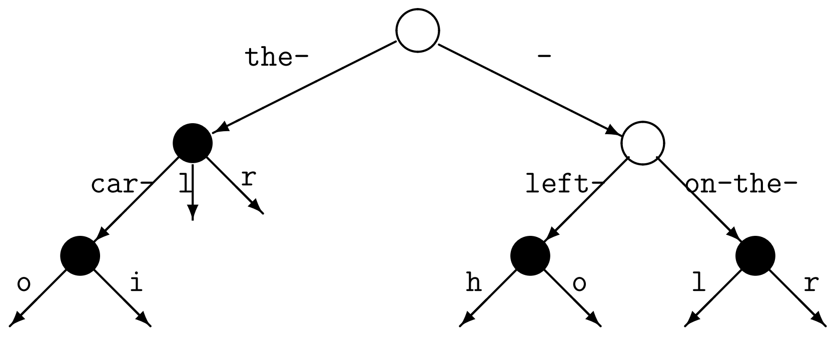

FIGURE 2: *Part of the compacted position tree*

Compacted position trees can be constructed in time linear in the length of the input text T. Also in linear time one could add to each vertex in the tree, the number of times the corresponding string occurs in T [2]. Thus, the set of vertices V and their labels can be constructed in time $O(n)$.

As to the edges, it suffices to check for each string x corresponding to a vertex v_x whether there should be a directed edge from it to a vertex corresponding to a string y. This will be the case if and only if the super-string $\overline{xy}$ exists. We have thus to search the position tree for each of the $|x|-1$ proper suffixes of x. If the path in the position tree corresponding to one of these proper suffixes leads to an internal node t, there will be an edge from v_x to each of the vertices in the subtree rooted by t.

If there is an upper limit K, independent of the size n of the text, on the length of a string corresponding to a vertex, the number of edges in the graph G will be linear. Indeed, a proper suffix of length i of x can be extended to at most $|\Sigma|^{K-i}$ strings y such that $\overline{xy}$ exists, therefore the outdegree of x is bounded by $|\Sigma|^K$, which is a constant relative to n. Assuming the existence of such an upper bound K is not unrealistic for real-life applications, where most re-occurring strings will tend to be words or short phrases. Even if some long strings re-occur, e.g., runs of zeros or blanks, it will often be the case that a bound K exists for the lengths of all except a few strings, which will still yield a linear number of edges. The following Theorem shows that such a linear bound on the number of edges can not always be found.

Theorem 2. *There are input texts T for which the corresponding graph $G = (V, E)$ satisfies $|E| = \Theta(|V|^2)$.*

Proof: Recall that a DeBruijn sequence of order k is a binary string B_k of length 2^k, such that each binary string of length k occurs exactly once in B_k, when B_k is considered as a circular string (see [15, Section 1.4]). For example, B_3 could be 00111010.

Consider the text $T = CB_k$, where C is the suffix of length $k-1$ of B_k. Each of the 2^k possible binary strings of length k appears exactly once in CB_k, so no string of length longer than k appears more than once. Every binary sequence of length $k-1$ appears exactly twice in a DeBruijn sequence, and as sub-strings of different strings of length k, thus there is a vertex in G for each of the 2^{k-1} strings of length $k-1$. More generally, for $2 \leq i < k$ (recall that strings of length 1 do not generate vertices), there are 2^i strings of length i that occur more than once in T, and each of these strings of length $< k$ is a substring of more than one string of length k. The number of vertices in the graph is therefore $\sum_{i=2}^{k-1} 2^i = 2^k - 4$.

Consider one of the strings x, corresponding to a vertex v_x, and denote its rightmost bit by b. Then there must be edges from v_x to at least all the vertices corresponding to strings starting with b. There are 2^{i-1} strings of length i starting with b, $2 \leq i < k$. Thus the number of edges emanating from v_x is at least $\sum_{i=2}^{k-1} 2^{i-1} = 2^{k-1} - 2$. Therefore the total number of edges in the graph is at least $(2^k - 4)(2^{k-1} - 2)$, which is quadratic in the number of vertices. ∎

3.2 Heuristics for Finding a Good Sub-graph

The optimization problem has been shown above to be NP-complete, so there is probably no algorithm to find an optimal solution in reasonable time. A family of simple greedy heuristics that have previously been used (see, e.g., [11]) includes the following steps:

```
1.      Decide on a set $S$ of potential strings and
                    calculate their weights $w(x)$ for $x \in S$
2.      Sort the set $S$ by decreasing values of $w(x)$
3.      Build the set $\mathcal{A}$ of the strings forming the meta-alphabet by
3.1         start with $\mathcal{A}$ empty
3.2         repeat until $|\mathcal{A}|$ is large enough or $S$ is exhausted
3.2.1           $x \longleftarrow$ next string of sorted sequence $S$
3.2.2           if $x$ does not overlap with any string in $\mathcal{A}$, add it to $\mathcal{A}$
```

The set S could be chosen as the set of all sub-strings of the text of length up to some fixed constant k, or it might be generated iteratively, e.g., starting with character pairs, purging the overlaps, extending the remaining pairs to triplets, purging overlaps again, etc. Such a strategy of not allowing any overlaps corresponds in our approach above to choosing an independent set of vertices.

To extend this greedy heuristic to include also overlapping strings, we have to update the weights constantly. It is therefore not possible to sort the strings by weight beforehand, and the data structure to be used is a *heap*. This will give us at any stage access in time $O(1)$ to the element x with largest weight $W(x)$, which should represent the expected additional savings we incur by adding x to the set $\mathcal{A}$, the currently defined meta-alphabet. $W(x)$ is therefore $w(x)$ if none of the neighbors of x belongs to $\mathcal{A}$, and it is more generally $w(x) - \sum_{y\in\mathcal{A}} w(x,y)$. The proposed heuristic is thus as follows:

```
1.         Define the set $S$ as strings corresponding to the
                               internal nodes of the compacted position tree
2.         for each $x \in S$ define $W(x) \longleftarrow w(x)$
3.         build heap of elements $W(x)$, with root pointing to maximal element
4.         Build the set $\mathcal{A}$ of the strings forming the meta-alphabet by
4.1            start with $\mathcal{A}$ empty
4.2            repeat until $|\mathcal{A}|$ is large enough or heap is empty
4.2.1              $x \longleftarrow$ string with weight at root of heap
4.2.2              add $x$ to $\mathcal{A}$ and remove $W(x)$ from heap
4.2.3              repeat for all neighbors $y$ of $x$ in graph
4.2.3.1                $W(y) \longleftarrow W(y) - w(x,y)$
4.2.3.2                if   $W(y) \le 0$        remove $W(y)$ from heap
4.2.3.3                else                 relocate $W(y)$ in heap
```

Note that $W(x)$ may indeed become negative, as can be seen when working through the example of Figure 1. This would mean that the potential gain obtained from including x in $\mathcal{A}$ might be canceled because of the overlaps.

The complexity of the heuristic can be evaluated as follows: step 1. can be done in time linear in the size n of the text as explained above. Let $m = |S|$ be the number of vertices. Steps 2. and 3. are $O(m)$. Step 4.2.1 is $O(1)$ and step 4.2.2 is $O(\log m)$. Any edge (x, y) of the graph is inspected at most once in the loop of 4.2.3, and for each such edge, a value $W(y)$ is either removed or relocated in the heap in time $O(\log m)$. Thus the total time of step 4. is $O(|E| \log m)$. As we saw in Theorem 2, $|E|$ could be quadratic in m. But when $W(y)$ is updated, it may only decrease, so that its relocation in the heap can only be to a lower level. The total number of updates for any $W(y)$ is therefore bounded by the depth of the heap, which implies that the complexity of step 4. is only $O(m \log m)$.

The Off-line heuristic of [4] has some similarities with the latter procedure, but differs in various aspects. It also successively chooses new meta-characters to be substituted, but it considers at each stage non-overlapping *occurrences* of a given meta-character within the text, whereas the new heuristic above considers overlaps between different meta-characters rather than between their occurrences. That is, Off-line works directly on the text itself, while the new heuristic tries to extract the meta-alphabet from the given graph, which models the text. The advantage of Off-line is then the fact that its decisions are based on real data and not on estimates, which yields very good compression performance, as reported in [4]. The advantage of the present approach, on the other hand, is in its focus on the model alone; this allows an independent subsequent application of different encoding schemes.

4 Extensions

4.1 Variable Length Encodings

In our above description, we have made various simplifying assumptions. In a more precise analysis, we shall try in a second stage to adapt the general strategy to more complex — and more realistic — settings.

When defining the graph, we assumed that the elements of the meta-alphabet are encoded by a fixed length code. Such a code will, however, be optimal only if the occurrence distribution of the elements is close to uniform. Otherwise, variable-length codes such as Huffman or arithmetic codes should be used. The problem is then one of the correct definition of the graph weights, but the generalization of the above method is not straightforward. We look for a set of strings which are selected on the basis of the lengths of their encodings; the lengths depend on their probabilities; these in turn are a function of the full set of strings, which is the set we wish to define.

A possible solution to this chicken and egg problem is as follows. We first estimate the average length, $\hat{\ell}$, of the strings $s_1, s_2, \ldots$ that will ultimately be selected. This can be done by some rough rule of thumb or by applying the heuristic iteratively. Clearly, if $n = |T|$ is the length of the text and N is the number of the selected strings, we have

$$n = \sum_{i=1}^{N} |s_i| freq(s_i).$$

Replacing now all $|s_i|$ by their estimated average value $\hat{\ell}$, we get

$$n = \hat{\ell} \sum_{i=1}^{N} freq(s_i),$$

from which an estimate for the total frequency, $W = \sum_{i=1}^{N} freq(s_i)$, of the selected elements can be derived as

$$W = \frac{n}{\hat{\ell}}.$$

Hence, when defining the probabilities in the weights of the vertices and edges, we shall use as approximation the frequency of a string divided by W, even though this is not a real probability distribution, as the selected values will not necessarily add up to 1. But the estimation bias is alleviated by the fact that the probabilities are only needed to determine the lengths of the codewords. We shall use $-\log_2 p$ as approximation for the length of a codeword that appears with probability p. This is exact for arithmetic coding, and generally close for Huffman coding [27].

The new weights are not measured in number of characters, but in number of bits. For a string $x = x_1 \cdots x_r$, the weight of the corresponding vertex will be

$$w(x) = freq(x) \left(- \sum_{i=1}^{r} \log_2 \frac{freq(x_i)}{W} + \log_2 \frac{freq(x)}{W} \right),$$

where $freq(x_i)$ are the frequencies of the individual characters making up the string, and similarly for the weights $w(x, y)$ of the edges. This is again an approximation, as we compare the cost of using the string as a single element versus the cost of using the constituent characters individually. So the string `the` would be compared with the characters `t`, `h` and `e`, whereas in fact the alternative to `the` could be using one of the *substrings* `th` or `he`. Once the set of meta-characters $\mathcal{A}$ is determined by the heuristic, we can update our estimate for the average length $\hat{\ell}$, and repeat the process iteratively. Even without a convergence guarantee, this will be useful for the elimination of bad strings.

4.2 Markov process

Many authors have commented on the importance of source modeling for good compression (see, e.g., [29]). We shall thus try to adapt the above techniques also to more involved models. The model suggested in [9] is based on extending first the alphabet, and then considering the sequence of meta-characters as generated by a first-order Markov process; each element in the sequence is then Huffman encoded according to its predecessor. The large overhead of the Markov process can be dealt with by clustering [10] and by using canonical Huffman codes [22]. This approach yields, on large textual test files, compression factors that compete well, and sometimes even beat, those of the best popular compression methods such as `gzip` or PPM, even though only very simple heuristics have been used for alphabet extension.

The problem in applying a Markov process to the definition of our graph is that the weights of the vertices are not fixed any more, and can not even be approached as done above for the variable length encoding, because the expected savings at any vertex now depend on the edge through which the vertex is accessed. The extension of our graph-based approach to deal with a Markov model will be deferred to future work, but we bring in the experimental section results of applying Markov based Huffman codes on the set of meta-characters generated by the previous methods.

4.3 Non-greedy parsing

A similar problem to the latter is encountered when we abandon the assumption that, once the meta-alphabet $\mathcal{A}$ is given, the text will be parsed greedily, i.e., by trying at each point to match the longest possible prefix of the remaining text with one of the meta-characters. If the weights are known beforehand, an optimal parsing can be found by a reduction to a shortest path problem [23].

Here again, we took a practical approach: instead of trying to generate a set $\mathcal{A}$ which should be optimal under the constraint of optimal parsing, the processes of alphabet construction and of the actual compression are separated. Once the set $\mathcal{A}$ is obtained by the above methods and the length of the encoding of each element can be evaluated, $\mathcal{A}$ is considered as *fixed*, and the optimal parsing for the *given* weights is generated.

5 Experimental Results

Papers suggesting new compression methods often present comparative performance charts on a large set of “standard” files like the Calgary or Canterbury corpora [1]. Since the purpose of this work is not the presentation of a specific heuristic, but rather of a general method for the improvement of static compression schemes, we restrict the experiments to only a few representative examples. Three texts of varying lengths and different languages were chosen for the tests: the King James version of the *Bible* in English, a French text by Voltaire called *Dictionaire philosophique* and a lisp program *progl* from the Calgary corpus. Table 1 lists the full sizes of these files in Mbytes, as well as the number of vertices in the graph, as percentage of the number of leaves of the compacted position tree.

	Bible	*Voltaire*	*progl*
Full size	3.32	0.53	0.068
# vertices	55%	53%	64%

TABLE 1: *Test file statistics*

It should be noted that the final heuristic is a product of several layers of approximations: the vertices of the graph do not cover all possible substrings, the weights describe the gains and losses only under certain constraints, and the heuristic does not necessarily find an optimal subset. It therefore made sense to relax the requirements at various stages of the construction, and for example not invest too much effort in the construction of the exact set of vertices, as many of them are ultimately discarded anyway.

As mentioned earlier, the separation of the model for the construction of the extended alphabet from the actual encoding applied to the elements of this alphabet, allows an independent evaluation of the performance of the different models. In the first set of experiments, we considered the weights in the graph corresponding to fixed length encodings and generated the set of meta-characters according to the second heuristic of Section 3.2. The meta-alphabet consisted of the basic 256 ASCII characters, to which 256 more strings have been adjoined, so that any meta-character could be encoded by 9 bits. The first line of Table 2 gives the sizes of these fixed-length encoded files, yielding a reduction of 40–60%.

Fixed length	*Bible*	*Voltaire*	*progl*
Best strings for fixed	1.97	0.29	0.031
Best strings for variable	2.20	0.32	0.034
Most frequent words	2.39	0.40	0.048
Most frequent pairs	2.02	0.31	0.042

TABLE 2: *Compression results for fixed-length encoding*

The meta-alphabet used to produce the next line of the table is based on the weights for the variable-length codewords of Section 4.1, but on which fixed-length encoding was applied. As expected, the compression results are inferior to those of the previous meta-alphabet.

To compare our method also with some simple techniques that are often used, we produced, for each file, a list of the most frequent words, as well as a list of the most frequent character bigrams. Each of these in turn were used to define a new set of meta-characters, again extending the basic 256 elements by 256 more. The last two lines of Table 2 refer to these meta-alphabets, which gave lower savings.

Variable length	*Bible*	*Voltaire*	*progl*
Best strings for fixed	1.48	0.23	0.0214
Best strings for variable	1.47	0.22	0.0212
Most frequent words	1.50	0.24	0.028
Most frequent pairs	1.73	0.26	0.036

TABLE 3: *Compression results with Huffman coding*

Table 3 are the corresponding results when Huffman coding is applied instead of fixed length coding. Obviously, all the values are smaller than those in the corresponding positions in Table 2. It can be seen that among the alternatives tested, the meta-characters produced by the heuristic on the graph with the weights corresponding to fixed length encoding indeed achieve the best compression by fixed length encodings, while the meta-characters produced with the weights of the variable length encoding are best when Huffman coding is applied. It is noteworthy that when passing from the fixed length to the variable length weights, the number of meta-characters in the parsing increases (this number is proportional to the size, since fixed length encoding is used), and nevertheless, the size of the corresponding Huffman encoded file is smaller. This is due to the fact that the distribution of frequencies in the latter case is much skewer than in the former, resulting in the overall gain. We also see that the differences between the different methods are smaller than for Table 2, as Huffman coding tends to partially correct the deficiencies of a bad choice of elements to be encoded.

The figures in Tables 2 and 3 are still far from the compression that can be achieved with adaptive dictionary methods. For instance, the Bible file can be reduced by `LZW` to 1.203 MB, by `gzip` to 1.022 MB, and by `bzip` to merely 0.74 MB. But recall that we are looking for a *static* method, which will allow random and not only sequential access to the compressed file, so that these dynamic methods are ruled out. On the other hand, using Huffman coding in combination with a first order Markov model as in Section 4.2, may achieve compression factors that can sometimes beat some of the dynamic methods.

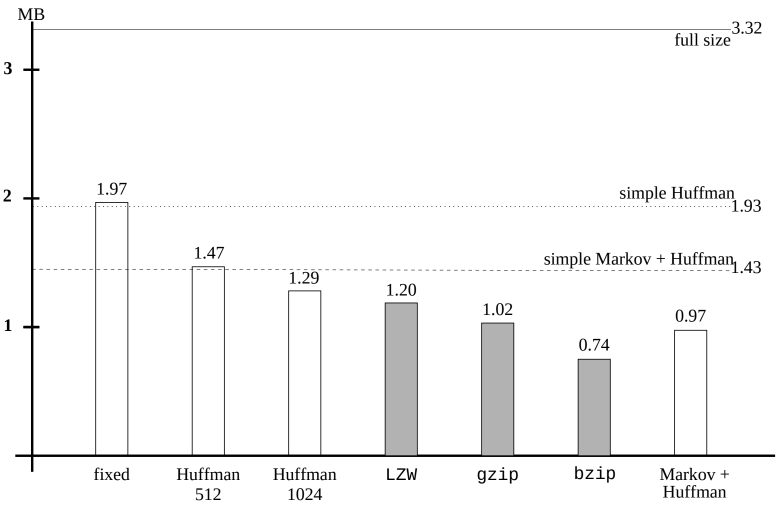

FIGURE 3: *Comparative chart of the compression of the Bible file*

Figure 3 schematically compares the various techniques on the Bible file. The upper horizontal line corresponds to the full size, and the dotted and dashed lines are produced by applying simple Huffman coding, and Huffman coding based on a first order Markov model, respectively, on a standard 256 character alphabet. The white histogram bars give the sizes of the compressed files when alphabet extension is used, and the grey bars bring the values of the dynamic methods. The leftmost two bars correspond to the values in Tables 1 and 2. By using a larger extended alphabet, these can be improved: with 1024 meta-characters (including the 256 basic ones), simple Huffman coding yields 1.29MB, which is only 8% more than for `LZW`. But with a Markov model, one can get even below one MB, which is better than `gzip` but still far from `bzip`; the value was obtained with an extended alphabet of 512 meta-characters, and includes 60K of overhead for the description of the model.

6 Concluding Remarks and Future Work

Alphabet extension is not new to data compression. However, the decision about the inclusion into the alphabet of various strings, such as frequent words, phrases or word fragments, has often been guided by some *ad-hoc* heuristic. The present work aims at making this decision in a systematic, theoretically justifiable way.

There are still many more details to be taken care of, in particular, devising more precise rules wherever approximations have been used. We have throughout applied greedy decisions to simplify processing. Though such greedy heuristics seem to give good results for the type of problem we consider here [20], we shall try in another line of investigation to adapt other approximation schemes to our case. The Independent Set optimization problem has been extensively studied, and some good heuristics exist (see [21]). In particular, we can use the fact that our graph has a low average vertex degree, and is of bounded vertex degree if we put some constraints on the lengths of the strings we consider. Another problem similar to ours, with edge weights but without weights on the vertices, has been approximated in [25].

Acknowledgement

As mentioned in the introduction, this research, as well as many other related ones, has been influenced by a long lasting connection with the Responsa Project, which has been headed for many years by Prof. Yaacov Choueka. I would like to take this opportunity of a volume in his honor to thank him for his help and guidance, which no doubt had a major impact on my research.

References

1. Arnold R., Bell T., A corpus for the evaluation of lossless compression algorithms, *Proc. Data Compression Conference DCC–97*, Snowbird, Utah (1997) 201–210.
2. Apostolico A., The myriad virtues of subword trees, *Combinatorial Algorithms on Words,* NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 85–96.

3. Apostolico A., Lonardi S., Some theory and practice of greedy off-line textual substitution, *Proc. Data Compression Conference DCC–98*, Snowbird, Utah (1998) 119–128.
4. Apostolico A., Lonardi S., Off-line compression by greedy textual substitution, *Proc. of the IEEE* **88** (2000) 1733–1744.
5. Aho A.V., Hopcroft J.E., Ullman J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA (1974).
6. Bell T.C., Cleary J.G., Witten I.A., *Text Compression*, Prentice Hall, Englewood Cliffs, NJ (1990).
7. Bell T., Witten I.H., Cleary J.G., Modeling for Text Compression, *ACM Computing Surveys* **21** (1989) 557–591.
8. Bentley J., McIlroy D., Data compression using long common strings, *Proc. Data Compression Conference, DCC–99*, Snowbird, Utah, (1999) 287–295.
9. Bookstein A., Klein S.T., Compression, Information Theory and Grammars: A Unified Approach, *ACM Trans. on Information Systems* **8** (1990) 27–49.
10. Bookstein A., Klein S.T., Raita T., An overhead reduction technique for mega-state compression schemes, *Information Processing & Management* **33** (1997) 745–760.
11. Bookstein A., Klein S.T., Ziff D.A., A systematic approach to compressing a full text retrieval system, *Information Processing & Management* **28** (1992) 795–806.
12. 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.
13. Cannane A., Williams H.E., General-purpose compression for efficient retrieval, *Journal of the ASIS* **52**(5) (2001) 430–437.
14. Choueka Y. Responsa: A full-text retrieval system with linguistic processing for a 65-million word corpus of jewish heritage in Hebrew, *IEEE Data Eng. Bull.* **14**(4) (1989) 22–31.
15. Even S., *Graph Algorithms*, Computer Science Press (1979).
16. Fraenkel A.S., All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16** (1976) 149–156.
17. Moffat A., Word-based text compression *Software – Practice & Experience* **19** (1989) 185–198.
18. Fraenkel A.S., Mor M., Perl Y., Is text compression by prefixes and suffixes practical? *Acta Informatica* **20** (1983) 371–389.
19. Garey M.R., Johnson D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco (1979).
20. Halldorsson M.M., Radhakrishnan J., Greed is good: approximating independent sets in sparse and bounded degree graphs, *Proc. 26th ACM-STOC* (1994) 439–448.
21. Hochbaum D.S., *Approximation Algorithms for NP-Hard Problems*, PWS Publishing Company, Boston (1997).
22. Klein S.T., Skeleton trees for the efficient decoding of Huffman encoded texts, in the *Special issue on Compression and Efficiency in Information Retrieval* of the *Kluwer Journal of Information Retrieval* **3** (2000) 7–23.
23. Klein S.T., Efficient optimal recompression, *The Computer Journal* **40** (1997) 117–126.

24. Klein S.T., Kopel Ben-Nissan M., On the Usefulness of Fibonacci Compression Codes, *The Computer Journal* **53** (2010) 701–716.
25. Kortsarz G., Peleg D., On choosing dense subgraphs, *Proc. 34th FOCS*, Palo-Alto, CA (1993) 692–701.
26. Larson N.J., Moffat A., Offline dicionary based compression, *Proceedings of the IEEE* **88**(11) (2000) 1722–1732.
27. Longo G., Galasso G., An application of informational divergence to Huffman codes, *IEEE Trans. on Inf. Th.* **IT–28** (1982) 36–43.
28. 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.
29. Rissanen J., Langdon G.G., Universal modeling and coding, *IEEE Trans. on Inf. Th.* **IT–27** (1981) 12–23.
30. Storer J.A., Szymanski, T.G., Data compression via textual substitution, *J. ACM* **29** (1982) 928–951.
31. Ukkonen E., On-line construction of suffix trees, *Algorithmica* **14**(3) (1995) 249–260.
32. Witten I.H., Moffat A., Bell T.C., *Managing Gigabytes: Compressing and Indexing Documents and Images,* Van Nostrand Reinhold, New York (1994).