The String-to-Dictionary Matching Problem*

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
Email: shapird@ash-college.ac.il

The String-to-Dictionary Matching Problem is defined, in which a string is searched for in all the possible concatenations of the elements of a given dictionary, with applications to compressed matching in variable to fixed length encodings, such as Tunstall's. Two algorithms based on suffix trees are suggested, the one focusing on the dictionary, the other on the pattern to be searched for. The problem is then extended to deal also with patterns that include gaps. Experiments on natural language text suggest that compressed search might use less comparisons for long enough patterns, in spite of a potentially large number of encodings.

Keywords: Compressed Matching, Tunstall, Suffix Trees

Received ; revised

1. INTRODUCTION AND BACKGROUND

Traditional research in *Pattern Matching* algorithms originally started from the simple question of how to locate a string $P = p_1p_2\cdots p_m$ within a text $T = t_1t_2\cdots t_n$, but then extended to various variants including locating sets of patterns, dealing with dictionaries and indexes, and many others. The techniques used to solve these problems sometimes rely on preprocessing the pattern(s), or the text, or both. In this sense, the *String-to-Dictionary Matching Problem* (SDMP), to be defined below, is yet another pattern matching scenario that, given its useful applications, should be considered.

A different line of thought gets to the SDMP by studying tradeoffs between several *Data compression* techniques. Classical Huffman coding [14] is optimal, but only if the set of elements to be encoded is fixed and codeword lengths are constrained to be integers. However, the compression ratio obtained by applying Huffman coding to a simplistic model of encoding the individual characters of a text independently is rather poor. Much better compression can be achieved by encoding the different *words* rather than the characters, at the price of dealing with a huge Huffman tree [20], but this might be a reasonable overhead worth paying, as the set of different words might be useful anyway, for example in large Information Retrieval systems.

The compression ratio is, however, not the only criterion by which data compression techniques should be judged. For instance, encoding and decoding speed are important parameters in many applications, and the variable length nature of Huffman codewords, which are not necessarily byte-aligned, can put a serious burden on the processing procedures. One way to alleviate the problem is to replace the binary Huffman codes by a 2^d-ary variant, in which the lengths of all the codewords are multiples of d, so choosing $d = 8$ results in all the codewords consisting of an integral number of bytes [21]. While the loss in compression might be significant for small alphabets, it is only of the order of a few percent when the basic elements to be encoded are words rather than characters.

Yet another feature of variable length codes which should be taken into account is to support searches directly within the compressed text, without having to decompress first. Denote the encoding and decoding functions by $\mathcal{E}$ and $\mathcal{D}$, respectively. Then supposing that one is given a compressed text $\mathcal{E}(T)$, instead of looking for a string P in the decompressed text $\mathcal{D}(\mathcal{E}(T))$, one may encode the pattern P and thus search for $\mathcal{E}(P)$ directly in $\mathcal{E}(T)$. This paradigm of *compressed matching* has become a research topic for its own sake in recent years. Searching directly in a Huffman encoded text might be tricky, because the occurrence of the binary string $\mathcal{E}(P)$ is not necessarily aligned on codeword boundaries in the compressed binary text $\mathcal{E}(T)$ [17]. A solution based on finite transducers has been suggested in [19]. In another approach, [21] propose to reserve 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

*This is an extended version of a paper that has been presented at the Data Compression Conference (DCC 2011), and appeared in its Proceedings, 143–152

256-ary to 128-ary. These *Tagged Huffman codes* have then been replaced by *End-Tagged Dense codes* (ETDC) in [6] and by (s, c)*-Dense codes* (SCDC) in [4]. An alternative code based on higher order Fibonacci numeration systems and yielding similar features is studied in [16].

The last three 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 to be encoded by non-increasing frequency and the fixed set of codewords by non-increasing length, and then match the two lists. Pushing the idea of the use of a fixed codeword set even further, one can revert back to *fixed-length* codes, but to avoid the total loss of any compression advantage, the elements to be encoded should remain of variable length. Such a method has been suggested by Tunstall [25] and some improvements are studied in [18].

Such variable-to-fixed length encodings are advantageous for fast decoding, but performing a compressed search is much more involved. The problem stems from the fact that the encoding of the pattern $\mathcal{E}(P)$ might on the one hand not be defined at all, and even if it is, this definition is not necessarily unique.

To illustrate this point on a small artificial running example, suppose the given set of variable length elements to be encoded, which we shall call the *dictionary*, consists of $D = \{$`A=aab`, `B=aba`, `C=abc`, `D=bcca`, `E=bc`, `F=bab`$\}$. The pattern to be sought for is $P =$ `abab`. If P itself were also an element of D, one might have looked for the (fixed-length) codeword $\mathcal{E}(P)$ in the compressed text. But this would not have been the only solution. The pattern P appears also as substring of several concatenations of codewords, in our case in `AB`, `AC`, `BD`, `BE`, `BF`, `DF`, `FB`, and `FC`. In `BF` the pattern appears even twice. Finding all the occurrences of P would thus involve generating first all the relevant concatenations, and subsequently searching for the encoded form of each of them in the compressed text.

This problem is reminiscent of the one treated in [9] in which a pattern is sought in a text generated by Straight-Line Programs (SLP). Their solution processes the production rules of the SLP, whereas the solution we suggest below deals directly with the elements of the given dictionary.

The brute force approach of generating all the potential concatenations is obviously not always feasible as their number might be exponential in the number of dictionary elements. Special cases of the SDMP, restricting the problem to searches for single words or phrases, are treated in [5, 3]. We suggest here solutions to the more general problem. In the next section, we formally define the SDMP and then suggest ways to solve it in Sections 3 and 4. In the first approach, the dictionary is stored in an extended trie which is traversed as guided by the pattern; in the second, it is the pattern that is pre-processed into a suffix tree, and the dictionary elements are used to traverse the tree.

One of the generalizations of the standard pattern matching problem is to patterns that might only be partially defined, for example, patterns with *gaps*, or equivalently, wildcards or don't care symbols [11, 12]. This has applications to music Information Retrieval [8], Molecular Biology [22, 23] and data compression [1]. A similar problem in an Information Retrieval (IR) context is known as dealing with *truncated terms* [2]. The extension of SDMP to pattern matching with gaps is studied in Section 5.

Finally, in Section 6 we consider an application of SDMP to the compressed matching problem on variable to fixed length encodings, such as Tunstall's.

2. DEFINITION OF THE STRING-TO-DICTIONARY MATCHING PROBLEM

Given is a dictionary of k variable length strings $D = \{d_1, \ldots, d_k\}$, where $d_i = x_{i,1}x_{i,2}\cdots x_{i,s_i}$, and all the characters $x_{i,j}$, $1 \le i \le k, 1 \le j \le s_i$ belong to some fixed alphabet Σ. Further given is a pattern $P = p_1p_2\cdots p_m$, which is also a character string over the same alphabet Σ. The String-to-Dictionary Matching Problem looks for occurrences of P in any string which can be obtained by concatenating, in any order and possibly with repetitions, elements of D.

More formally, a solution to the SDMP consists of a pair of elements

$$\langle j_1, j_2, \ldots, j_r \ ; \ q\rangle, \tag{1}$$

the first of which is a sequence of r not necessarily different indices $j_1, j_2, \ldots, j_r \in \{1, 2, \ldots, k\}$, and the second being a starting position $q \le s_{j_1}$, such that either:

1. $r = 1$ and the pattern P is entirely contained within one of the dictionary strings, starting from position q; or,
2. $r > 1$, and P can be parsed as the suffix starting at position q of d_{j_1}, followed by occurrences of the $r-2$ elements $d_{j_2}, \ldots, d_{j_{r-1}}$, followed by a prefix of d_{j_r}, as schematically represented in Figure 1.

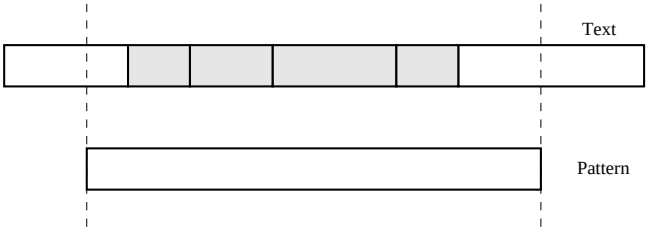

FIGURE 1. *Schematic representation of a solution to the SDMP*

The problem is to find all the solutions, and its complexity depends on the relationships between m, the

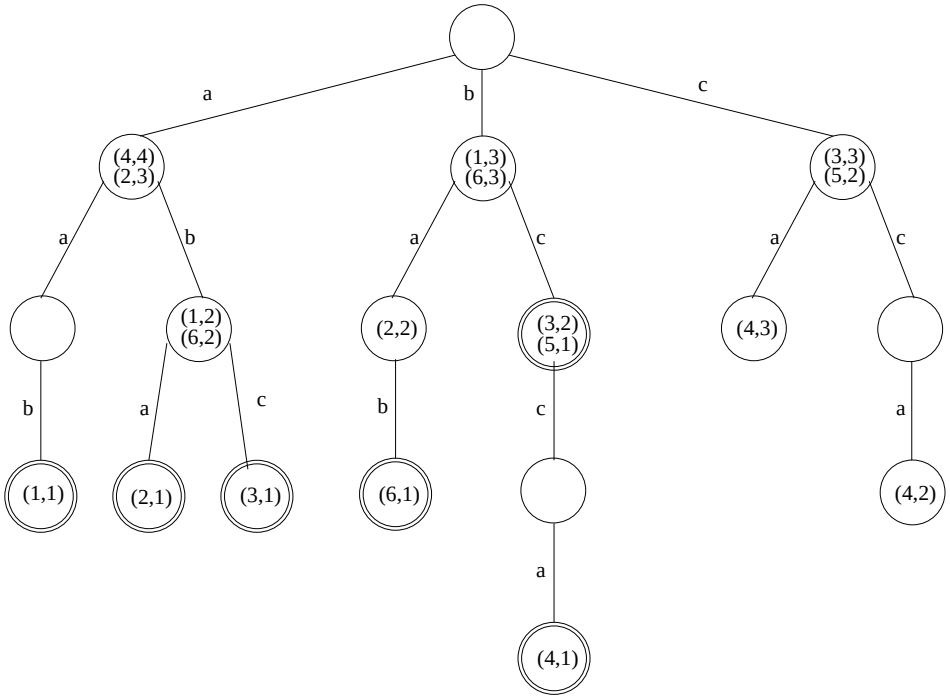

FIGURE 2. *Extended trie for* $D = \{\texttt{aab}, \texttt{aba}, \texttt{abc}, \texttt{bcca}, \texttt{bc}, \texttt{bab}\}$

size of the pattern, and the lengths s_i of the elements of the dictionary. If m is relatively small and the s_i are large, in particular, if $\min_{1\le i\le k} s_i \ge m$, then P can only appear in its entirety within a single element (case 1 above), or as a substring of the concatenation of exactly two elements (case 2 above, with $r = 2$), having a non-empty overlap with each of them. There are thus at most k elements and k^2 pairs of elements to scan, and a straightforward approach might then be successful.

For larger m, however, the number r of potential elements which have to be concatenated increases, and the number of possible options might grow exponentially as k^r. Rather than exhaustively generating all the possible combinations, we shall propose an approach based on the idea of *suffix trees*.

3. GENERATING ALL THE SOLUTIONS OF THE SDMP

Start by generating all the suffixes of all the elements in D, that is, consider the set $\mathcal{S}$ of strings

$$\mathcal{S} = \{d_{i,j} = x_{i,j}x_{i,j+1}\cdots x_{i,s_i}, \quad 1 \le j \le s_i, 1 \le i \le k\}.$$

These $O(\sum_{i=1}^{k} s_i^2)$ strings 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 a string T would be the trie for which the set of strings associated to its leaves is the set of the suffixes of T. In our case we define an *extended trie*, and the strings used to build it are the elements of $\mathcal{S}$, but departing from the convention for standard suffix trees, the elements will not correspond to the labels of the leaves only.

In fact, the same string may appear as suffix of more than one element of the dictionary, and a suffix of one element can be the prefix of another. We shall thus retain from the suffix tree procedures only the way of traversing the trie, but devise another labeling scheme, which is adapted to our SDMP.

If a node v of the trie is associated with one of the strings $d_{i,j}$ of $\mathcal{S}$, a pointer of the form (i, j) will be stored as (a part of the) label of v. A given string may appear more than once in $\mathcal{S}$, so that the label of v may consist of several pairs. Returning to our running example and renaming the elements $\texttt{U}, \texttt{V}, \ldots, \texttt{Z}$ by $d_1, \ldots, d_6$, respectively, Figure 2 shows the resulting extended trie. Note that there are three kinds of nodes: those that are non-labeled, those labeled only by proper suffixes of elements of D, and those labeled (possibly among others) by entire elements of D. The latter are indicated by double circles in Figure 2.

To find all the solutions to the problem, the pattern P will be used to traverse the trie, possibly several times. An *advancing step* from a current node v in the trie

according to x is defined as selecting the edge (v, w) emanating from v and labeled x, if there is such an edge, and setting the new current node to be w. The proposed procedure starts with the current node set as the root of the trie and repeatedly performs advancing steps according to the characters of the pattern. Reaching a labeled node means that the end of an element has been detected. If this happens for the first time, that is, during the first traversal of the trie, then we are looking also for proper suffixes of the elements in D, so any labeled node should be handled. For subsequent traversals of the trie, we are only interested in detecting entire elements (those which appear in grey in Figure 1), or, if this is the last iteration, a prefix of an element. The way to distinguish these cases is by looking at the second parameter of the label (i, j): if $j = 1$, the string associated with the current node is an entire element of D.

The formal algorithm appears in Figure 3. It uses the global variables in_p and in_trie, the first giving the index of the current character in the pattern P, the second pointing to the current node in the trie. The fact that multiple overlapping solutions are possible is handled by means of a queue Q, in which partial solutions are temporarily stored. Inserting an element x into Q and extracting the next element from Q into x are denoted, respectively, by $Q \Longleftarrow x$ and $x \Longleftarrow Q$. The elements in the queue Q are triples of the form (Seq, q, ind), where Seq is a sequence of indices representing the beginning of a potential solution as defined in equation (1), q is the starting position of this partial solution, and ind is the index of the character within the pattern P from which the extension of the partial solution has to be checked.

The algorithm distinguishes between the first time the iteration is processed and the following iterations. This is done by a global variable named first which is initialized to TRUE but set to FALSE at the end of the first iteration. The first element, d_{j_1}, of which possibly only a proper suffix is matched, is dealt with in the first iteration. Subsequent iterations handle the other elements d_{j_i}, for $i > 1$. In the first iteration, whenever a node having *any* label is encountered, this means that (the suffix of) an element has been detected, so it is inserted into the queue for later processing (lines 14–15). In the following iterations, only if the label corresponds to an entire element, it will be concatenated at the end of the sequence detected so far and reinserted into the queue if the pattern is not yet exhausted (lines 16–17). Concatenation of strings A and B is denoted by $A \,||\, B$.

Lines 18–24 correspond to the case that the end of P has been reached. If at that moment the current node in the trie is a leaf, this is the special case in which d_{j_r}, the last element in the matching sequence, is entirely contained in P (in Figure 1, the grey elements extend to the right edge of P). If the current node v is an internal node, this is the case in which only a proper prefix of d_{j_r} is a suffix of P. One has then to consider all the nodes in the subtree rooted at v, and refer to their labels: in the first iteration, all labeled nodes are relevant, and this is the special case in which P is fully contained within a single element; in the following iterations, only doubly circled nodes should be considered — they correspond to all the possible extensions of the matching pattern.

The algorithm prints all the generated solutions, if there are any. At the end of the while loop starting at line 8, the pattern might not have been scanned completely yet, that is, $in\_p \leq m$. This is the case in which P does not appear in the current sequence of concatenated elements, and accordingly, nothing is printed.

As to the complexity of the algorithm, one could have thought that the number of times the pattern is scanned is bounded by the number of solutions, that is, the

```
1.  in_p  ⟵  1
2.  Q  ⟸  ∅
3.  first  ⟵  TRUE
4.  repeat
    {
5.      in_trie  ⟵  root
6.      if Q ≠ ∅ then
7.          (Seq, q0, in_p)  ⟸  Q
8.      while in_p ≤ m and there is an edge (in_trie, w)
                    emanating from in_trie labeled p_{in_p}
        {
9.          in_trie  ⟵  w
10.         in_p  ⟵  in_p + 1
            // reached end of element of D or of a prefix
11.         if in_trie is labeled then
            {
                // not all of P yet processed
12.             if in_p ≤ m then
13.                 for all labels (ℓ, q) of in_trie
14.                     if first is TRUE then
15.                         Q  ⟸  (ℓ, q, in_p)
                        // reached end of element of D
16.                     else if q = 1 then
17.                         Q  ⟸  (Seq || ℓ, q0, in_p)
            }
        }
        // reached end of P
18.     if in_p > m then
19.         scan sub-trie rooted at in_trie and
20.         for all labels (ℓ, q) of all nodes in the sub-trie
21.             if first is TRUE then
22.                 print ⟨ℓ ; q⟩
23.             else if q = 1 then
24.                 print ⟨Seq || ℓ ; q0⟩
25.     first  ⟵  FALSE
26. } until Q = ∅
```

FIGURE 3. Formal algorithm for SDMP

number of occurrences of P in different concatenations of elements of D. But this does not take all the work into account that is invested for finding partial solutions that ultimately do not lead to a full match. A correct bound on the complexity would rather be

$$\max_{1\le j\le m}\left[j\times\Big(\text{number of occurrences of } p_1\cdots p_j \text{ in } \bigcup_{\substack{i_1,\ldots,i_r\in\{1,\ldots,k\}\\ s_{i_1}+\cdots+s_{i_r}\ge m}} d_{i_1}d_{i_2}\cdots d_{i_r}\Big)\right],$$

which is linear in the number of characters in all the occurrences of the patterns or their prefixes. This is in contrast with generating all possible concatenations, which can be exponential even if the number of occurrences of the patterns is small.

We now consider a few examples of possible patterns:

1. P = `bccb` – there is no path in the suffix trie corresponding to P. In the first part, the triples (1,3,2), (6,3,2), (3,2,3) and (5,1,3) have been loaded into the queue, but all the matching attempts will fail and nothing is printed.
2. P = `cacc` – there are paths in the suffix trie corresponding to P, but no path ends at the root of a subtree including nodes that are doubly circled. Thus again, there is no output.
3. P = `caaba` – traversing the trie with P goes through the node corresponding to `c`, where (3,3,2) and (5,2,2) are inserted into Q, and through the node corresponding to `ca`, for which $Q \Longleftarrow (4,3,3)$. The algorithm then moves to the second phase, rescanning the trie starting from the root. (3,3,2) and then (5,2,2) are retrieved from Q, so the suffix `aaba` is used, and only complete words are considered. Thus, only the node labeled (1,1) is relevant, and the triples inserted into Q are ((3 || 1),3,5) and ((5 || 1),2,5). The next step is to retrieve (4,3,3) from Q; the suffix `aba` is used, and since only complete words are considered, the leaf labeled (2,1) is reached. Since all of P has been scanned, nothing is added to the queue, but at lines 23–25 the first solution $\langle 4,2\ ;\ 3\rangle$ is output. Finally, ((3 || 1),3,5) and then ((5 || 1),2,5) are retrieved from Q, indicating that the suffix to be used to guide the traversal of the trie is `a`. The subtree rooted at the node associated with `a` has three doubly circled nodes, causing the printing of $\langle 3,1,1\ ;\ 3\rangle$, $\langle 3,1,2\ ;\ 3\rangle$ and $\langle 3,1,3\ ;\ 3\rangle$, and then of $\langle 5,1,1\ ;\ 2\rangle$, $\langle 5,1,2\ ;\ 2\rangle$ and $\langle 5,1,3\ ;\ 2\rangle$.

4. ALTERNATIVE SOLUTION OF THE SDMP — INVERSE APPROACH

The approach of the previous section was to consider the dictionary D as fixed and thus to construct an extended trie based on its elements in a preprocessing stage. The trie is then accessed repeatedly as guided by the characters of the given pattern P. In certain applications it could be useful to inverse the approach and construct a suffix tree for P that will be accessed via the elements of the dictionary D. This will be true, e.g., if the pattern is short, or if a fixed pattern P is searched for in a large number of different dictionaries.

We start by constructing the suffix tree of $P = p_1\cdots p_m$, which is the trie for which the set of associated strings is $\{P_i = p_ip_{i+1}\cdots p_m \mid 1\le i\le m\}$, the set of suffixes of P. Note that since we do not extend the pattern with an appended \$, there is no one-to-one correspondence between suffixes of the pattern and leaves in the trie, and a suffix might be associated with an internal node. A node in the trie that is associated with a substring $p_i\cdots p_j$ of P, with $1\le i\le j\le m$, will be labeled by $[i-1,j]$. Since a given substring can appear more than once in P, certain nodes may have more than one label. As a running example, consider the last example pattern given earlier, P = `caaba`. Figure 4 shows the corresponding suffix tree and its labels.

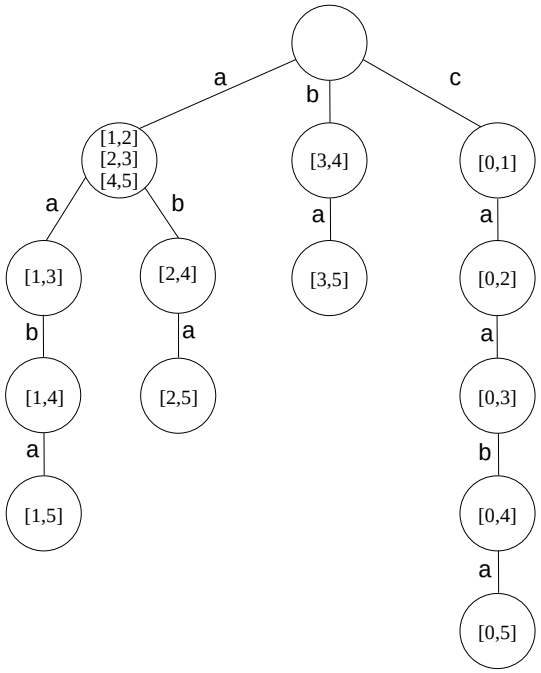

FIGURE 4. Suffix tree for pattern P = `caaba`

The idea behind the labeling scheme stems from reducing a parsing of the pattern P to a path in a corresponding graph $G_P = (V,E)$, as done, for example, in [15]. For a pattern of length m, there are $m+1$ nodes $V = \{0,1,2,\ldots,m\}$. We are then given a set of text fragments, in our case, the dictionary D. If one of its elements, d, is a substring $p_i\cdots p_j$ of P, then there will be an edge $[i-1,j]$ in E and the edge will be labeled by the name of the element, namely d. The nodes and solid arcs in Figure 5 show the graph corresponding to P =

caaba and the dictionary $D = \{$A,B,C,D,E,F$\} = \{$aab, aba, abc, bcca, bc, bab$\}$ used above. If there were a parsing of P into a sequence of dictionary elements $d_{j_1}, \ldots, d_{j_s}$, then there would be a path in G_P from 0 to m consisting of s edges and labeled, respectively, d_{j_1} to d_{j_s}. Our case is more complicated as we do not require the first and last elements of the parsing sequence to be entire dictionary elements. Rather, the first one can be the proper suffix and the last one a proper prefix of some of the d_i, as shown by the broken edges in Figure 5.

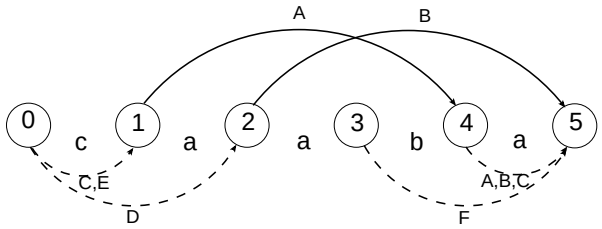

FIGURE 5. Graph for pattern $P =$ caaba and dictionary $\{$A,B,C,D,E,F$\} = \{$aab, aba, abc, bcca, bc, bab$\}$

The algorithm for SDMP is then the following. Build the suffix tree of P and start with a graph G_P without edges. Process all the dictionary elements d by traversing the trie from its root according to the characters in d. If this traversal succeeds and ends at some node labeled $[i-1, j]$, this means that d is a substring of P so the edge $[i-1, j]$ is adjoined to the graph. This is the case for aab and aba in our example. If the traversal does not succeed, as for abc, bcca, bc or bab, no action is taken.

So far, the construction has dealt only with dictionary elements that appear in their entirety in the pattern. It remains to deal with the prefixes and suffixes. Generate all the proper suffixes of all the dictionary elements and compare such a suffix of length i with the prefix of length i of P, $p_1 p_2 \cdots p_i$. In case of equality, add the edge $[0, i]$ and label it with the name of the corresponding element. Then generate all the proper prefixes and compare a prefix of length j with the suffix of length j of P, $p_{m-j+1} \cdots p_m$. If this succeeds, add the edge $[m-j, m]$ and label it accordingly. Note that an edge in G_P can get more than one label, as can be seen in the example for edges $[0, 1]$ and $[4, 5]$; in addition, the same dictionary word can label more than one edge in G_P, as can be seen in the example for words A, B, and C. What remains to do is generating all the paths from 0 to m in the graph G_P: the sequence of labels on these paths are the solutions of the SDMP.

An advantage of generating the solutions in this form is that overlapping solutions are compactly described. On our example, one of the paths gives the sequence C,E - A - A,B,C, which in fact is a shortcut for the set C-A-A, C-A-B, C-A-C, E-A-A, E-A-B, E-A-C.

The complexity of this variant is $O(m)$ for the construction of the suffix tree, plus $O(\sum_{i=1}^{k} s_i)$ for using all the dictionary elements to traverse the tree, plus $O(\sum_{i=1}^{k} s_i^2)$ for dealing with all prefixes and suffixes of the d_i. If the last term is dominant, it can be reduced by the following strategy. Consider, in the example above, the element bab. Using it for a traversal of the suffix tree is not successful, but one of the nodes visited during the traversal is labeled $[3, 5]$, indicating that a prefix of the dictionary term bab is a suffix of the pattern. The same is true for the element abc, during the traversal of which the algorithm passes through a node labeled $[4, 5]$. These examples may be generalized to amend the above algorithm to consider not only full traversals of the suffix tree, but also partial matches of prefixes of dictionary elements with suffixes of the pattern, indicated by nodes labeled $[i, m]$ for some i. This takes thus care of all the prefixes in a single scan of all the elements, using only $O(\sum_{i=1}^{k} s_i)$, and not $O(\sum_{i=1}^{k} s_i^2)$ steps. To deal also with all the suffixes, it suffices to construct a suffix tree of the reversed pattern, $P^R =$abaac on our example, and apply the above traversal algorithm with the set of reversed dictionary elements, {baa, aba, cba, accb, cb, bab} in our case. There are of course parts in this approach that overlap and could be omitted, but the overall complexity is bounded by $O(m + \sum_{i=1}^{k} s_i)$. In practice and for certain values of s_i, the approach without the reversed suffix tree might still be preferable, in spite of the larger upper bound on the complexity. This will be the case, e.g., if m is large or if many of the s_i are small. In particular, if even $\sum_{i=1}^{k} s_i^2$) is dominated by m, it might be wasteful to construct also a suffix tree for the reversed pattern.

5. MATCHING PATTERNS WITH GAPS

In this section we consider the extension of the SDMP to the problem of pattern matching with gaps. Let * stand for a variable length don't care sequence, one is interested in retrieving all the terms of a given dictionary of the form X*, *X, *X* or X*Y, where X and Y are some given strings. The problem in IR is to generate search terms which have to be extracted from some dictionary; if it is lexicographically sorted, the access to the set of strings matching a prefix truncated term like *X is not trivial. In pattern matching applications, however, prefix or suffix truncation is not an issue, since anyway only the pattern P itself is located, and not some element of a dictionary that contains P as a substring. The only relevant problem in this context is therefore what has been called *infix* truncation, of the form X*Y.

In practical applications, the length of the string matched by the don't care is often limited, otherwise the requested set of solutions of X*Y is just the set of non-overlapping pairs of the Cartesian product of the solution sets of X and Y, and between them the concatenation of dictionary words with lengths summing up to the number of don't cares. We shall denote by $*_n$ the don't care sequence of length at most

n. Alternatively, one might as well be interested in fixed length don't cares. Borrowing the notation used for *motifs* in [1], we define X_Y (X_nY) to stand for a string in which X and Y are separated by exactly one (exactly n) character(s).

We are interested in an algorithm that reports all concatenations of words of a given dictionary in which X_nY occurs, using the solutions for SDMP for X and Y independently, possibly by applying the algorithm given in Figure 3. First, the algorithm should generate the solution sequences Seq_x and Seq_y for X and Y, respectively. Three cases have to be dealt with, according to the length, rem_x, of the remaining (suffix) characters of Seq_x, plus the length $in_y - 1$ of the remaining (prefix) characters of Seq_y. Let us denote this sum of lengths by cov, which is the number of characters between X and Y not belonging to X or Y themselves but already covered by the dictionary elements which include X and Y in their concatenations. The cases are schematically illustrated in Figure 6.

```
1.  Q  ⟸  set of solutions of SDMP for X
2.  while Q ≠ ∅
3.      ⟨Seq_x ; in_x⟩  ⟸  Q
4.      ℓ  ⟵  last(Seq_x)
        // number of remaining chars in d_ℓ
5.      rem_x  ⟵  Σ_{t∈Seq_x} |d_t| − in_x + 1 − |X|
6.      for each solution ⟨Seq_y ; in_y⟩ of SDMP for Y
7.          f  ⟵  first(Seq_y)
            // in_y − 1: number of non-matching chars in d_f
8.          cov  ⟵  rem_x + in_y − 1
9.          if cov = n
10.             print ⟨Seq_x || Seq_y ; in_x⟩
11.         else if cov = |d_ℓ| + n and ℓ = f
12.             NSeq_y  ⟵  Seq_y
                           without first element f
13.             print ⟨Seq_x || NSeq_y ; in_x⟩
14.         else if cov < n
15.             L  ⟸  ∅
                // Λ is the empty string
16.             generate_concat(Λ, n − cov)
17.             for all elements ℓ of L
18.                 print ⟨Seq_x || ℓ || Seq_y ; in_x⟩

19. generate_concat(s, h)
20.     for all elements d_i of D for which |d_i| ≤ h
21.         if |d_i| = h
22.             add s || i to list L
23.         else
24.             generate_concat(s || i , h − |d_i|)
```

FIGURE 7. *SDMP with exactly n don't cares*

- If cov is **exactly** n, the solution for X_nY is just the concatenation of the partial solutions for X and Y (Case 1 in Figure 6).
- If cov is **more** than n, and the last word, d_ℓ, of Seq_x is identical to the first word of Seq_y, that is, the sequences of lengths rem_x and $in_y - 1$ partially overlap, then if there are exactly n characters left in the matching word, the last element of Seq_x and the first of Seq_y can be merged into a single occurrence in the sequence containing X_nY. This is verified by checking whether $(|d_\ell| - rem_x) + (|d_\ell| - in_y + 1) + n = |d_\ell|$, where the expressions in parentheses refer to the number of used characters in X and Y, respectively. This formula can be simplified to $cov = |d_\ell| + n$ (Case 2).
- If cov is **less** than n, it should be checked which words of the dictionary can be concatenated in between X and Y so that their lengths together with the remaining characters sum up to n (Case 3).

The formal algorithm is given in Figure 7. The generation, in Case 3, of the concatenations of all elements $d_{i_1}, \ldots, d_{i_r} \in D$ such that $i_1, \ldots, i_r \in \{1, \ldots, k\}$ and $|d_{i_1}| + \cdots + |d_{i_r}| = n - cov$ is done by means of the recursive procedure generate_concat which gets two parameters: a string s containing the partial sequence of the indices of already concatenated elements, and a number h, representing the length of characters still to be covered. When all the $n - cov$ characters are exhausted, the corresponding sequence of indices ℓ is adjoined to a list L. At the end, all possible concatenations of Seq_x and Seq_y with all elements ℓ of L between them are printed.

Note that the second argument of generate_concat is always strictly positive when it is invoked ($n - cov$ in line 16., and $h - |d_i|$ in line 24.), so there is no need to check for $h > 0$ within the procedure.

We now consider three examples which illustrate the above cases. Returning to our previous example, we use the same dictionary of Figure 2, and assume that for all cases X is caaba with output $\langle 3, 1, 1 \ ; \ 3\rangle$, $\langle 3, 1, 2 \ ; \ 3\rangle$, $\langle 3, 1, 3 \ ; \ 3\rangle$, $\langle 4, 2 \ ; \ 3\rangle$, $\langle 5, 1, 1 \ ; \ 2\rangle$, $\langle 5, 1, 2 \ ; \ 2\rangle$ and $\langle 5, 1, 3 \ ; \ 2\rangle$. For our examples we also consider Y to be bba with output $\langle 1, 6 \ ; \ 3\rangle$ and $\langle 6, 6 \ ; \ 3\rangle$.

1. Case 1: Assume that $n = 4$, and let us refer to the triplets $\langle 3, 1, 1 \ ; \ 3\rangle$ and $\langle 1, 6 \ ; \ 3\rangle$, which contain occurrences of X and Y, respectively. In this case $cov = 2 + 3 - 1 = 4$ which is equal to n, and therefore the sequences are concatenated and $\langle 3, 1, 1, 1, 6 \ ; \ 3\rangle$ is output, referring to the sequence ab**c-aab-a**ab**-aa**b**-ba**b, (X and Y are given in bold).
2. Case 2: Now let us assume that $n = 1$ and refer to the same triplets as in Case 1. In this case $cov = 2+3-1 = 4$ is more than n and the sequence for X ends with the word d_1 =aab which is the same word that opens the sequence for Y. Since

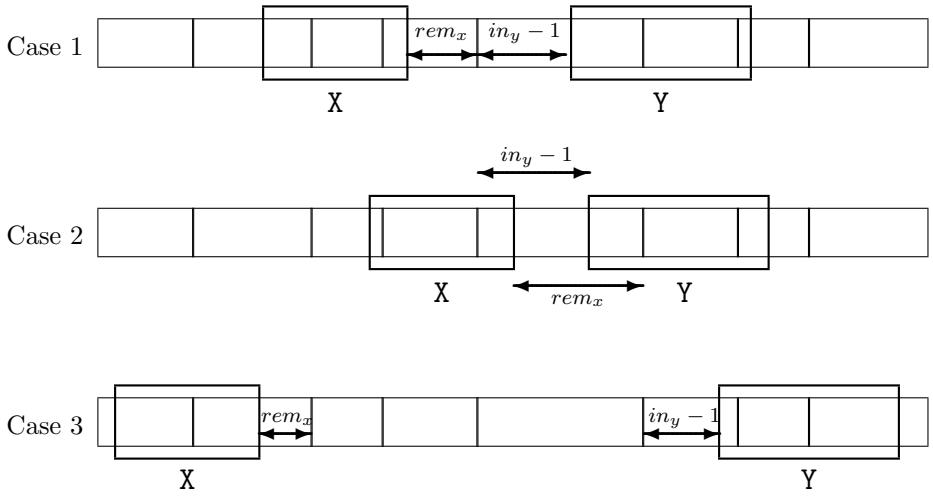

FIGURE 6. Illustration of the different cases for SDMP with fixed length gaps

$cov = 2 + 3 - 1 = 3 + 1 = |\texttt{aab}| + n$ the sequence $\langle 3, 1, 1, 6\ ;\ 3\rangle$ is output, referring to the sequence **ab**c-**aab**-a**ab**-**ba**b.

3. Case 3: For the case where n is equal to 6 which is greater than the number of remaining characters, the only word with length less or equal to $6 - 4 = 2$ is d_5 =bc. This will cause the printing of $\langle 3, 1, 1, 5, 1, 6\ ;\ 3\rangle$, referring to the sequence **ab**c-**aab**-a**ab**-bc-a**ab**-**ba**b.

6. APPLICATION TO COMPRESSED MATCHING IN TUNSTALL ENCODED TEXTS

One of the motivations of this work mentioned above is to enable a search directly in a text that has been compressed by some variable-to-fixed length encoding scheme, such as Tunstall's. Given a dictionary D and a pattern P to be searched for, one has first to generate all the minimal superstrings of P that can be obtained by concatenating elements of D; in terms of sections 3 and 4 above, these are all the solutions to the SDMP with D and P as parameters. The problem is that the search has then to be performed for each of the solutions, but their number might be much too large. If the purpose of the compressed search was to save the decompression time, the savings should obviously not be wasted by performing too many search iterations.

Empirical tests on natural language texts showed that, nevertheless, it is possible to search for all the solutions, because they are largely overlapping. The following example illustrates this point. The sample text used was the King James version of the English Bible, which has been stripped of all punctuation signs, leaving only blank, visualized as a dash -, and upper and lower case letters. The variable to fixed length codes were generated according to Tunstall's algorithm with 12-bit codewords, which produced a dictionary D of 4081 elements. For the pattern o-the-sum-, the number of combinations of codewords in which it appeared was 58220, but most of them were formed by the same elements: -th, e-s and um. The missing prefix, o, is the suffix of 76 elements of D, and the missing suffix, -, is the prefix of 766 elements of D, which already accounts for 58216 combinations, all of which have the same three elements (which correspond to the grey elements in Figure 1) in their middle part.

For longer patterns, this suggests the following strategy, based on an idea mentioned in [24] to improve the Boyer-Moore (BM) [7] algorithm: if there are significant fluctuations in the probabilities of the characters of the underlying alphabet, do not scan the pattern necessarily from its end towards its beginning (right to left), but let the scan order depend on the probability of occurrence, trying to match the rare characters first. The likeliness of an earlier mismatch is thereby increased, and so is the expected number of characters the pattern might be shifted in each iteration. For example, in a long enough pattern, if the last character has high probability but the next to last is rare, it may be advantageous to just ignore the last character and start the matching process from the penultimate position.

In our case, we have to look for a large set of patterns, almost all consisting of the same sequence of characters, except the first and the last. The idea is thus to eliminate these extreme elements in all the solutions and to search, in terms of the notation of Section 2, for $d_{j_2}, \ldots, d_{j_{r-1}}$ for each solution. In practice, this will reduce the potentially very large number of patterns to just a few, and even these may have extensively overlapping suffixes, suggesting to use the Commentz-Walter (CW) algorithm [10] to search for all of them at once.

Figure 8 shows the results of the following test. The test patterns have been chosen originating at

regular intervals in the Bible, more precisely at positions $500000i$, for $i = 1, \ldots, 6$, and for each starting position, the patterns were the substrings of lengths $10, 20, \ldots, 100$. The text has then been encoded using 12-bit and 16-bit Tunstall codes, the most frequent elements of which were: {`er`, `ho`, `oh`, `-u`, `u-`, `--e`, `-e-`,...}, and {`rae`, `rea`, `aho`, `aoh`, `hao`, `hoa`,...}, respectively. Searching for the patterns using any of the above algorithms gave tens of thousand solutions for 12-bit, and hundreds of thousand for 16-bit Tunstall, but in all cases, stripping the first and last elements reduced the size of the solution sets to only 1 to 4.

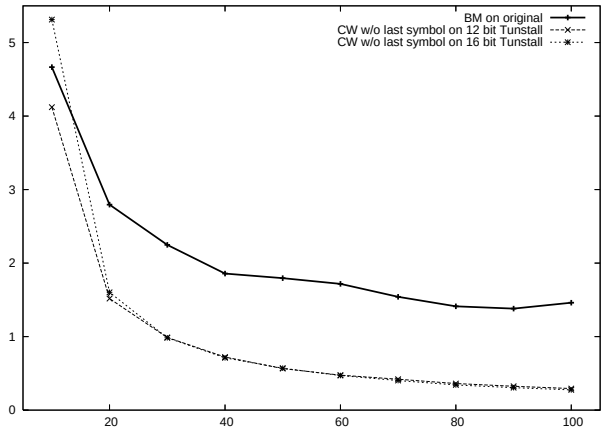

FIGURE 8. Expected number of comparisons as a function of pattern length

Horspool's variant [13], using only the Δ_1 function of the BM algorithm was used for a CW search of the set of solutions corresponding to each pattern. Since the patterns belonging to the same set of solutions are not necessarily of the same length and CW aligns them by their right end, it is convenient to change the notation so that the index of the last character in each pattern is m. The set of t patterns to be searched for will thus be $\{P^{(1)}, \ldots, P^{(t)}\}$, where $P^{(i)} = p^{(i)}_{\ell_i} p^{(i)}_{\ell_i+1} \cdots p^{(i)}_m$.

The expected number of positions the text pointer can be shifted after each mismatch is

$$sh = \sum_{x \in \Sigma} Pr(x) \min_{1 \le j \le t} \Delta_1(P^{(j)}, x),$$

where Σ is the alphabet (in our case, a 12-bit or 16-bit Tunstall code), $Pr(x)$ is the probability of occurrence of x in the text and $\Delta_1(P^{(j)}, x)$ is the number of positions the pattern can be shifted if the last character of $P^{(j)}$ is mismatched with character x. The expected number of comparisons between shifts is

$$cmp = \sum_{i=\max\{\ell_1,\ldots,\ell_t\}}^{m} \big[(m-i+1)(1-\mathsf{Prob}(i)) \prod_{j=i+1}^{m} \mathsf{Prob}(j)\big],$$

where

$$\mathsf{Prob}(i) = \sum_{x \in \{p_i^{(1)},\, p_i^{(2)},\, \ldots\, ,\, p_i^{(t)}\}} Pr(x)$$

is the sum of the probabilities of the different elements in the ith position of the t patterns. In particular, if the element in a given position i is the same for all t patterns, $\mathsf{Prob}(i)$ will just be $Pr(p_i^{(1)})$.

The overall expected number is approximated by $(n/sh) \times cmp$, where n is the size of the text. This is an approximation for several reasons: (i) it assumes independence of the characters, (ii) it truncates the patterns to be all of the same length, (iii) it does not use the Δ_2 function of the BM algorithm. For long enough patterns and for sets of patterns with extended overlaps at their suffixes, as in our application, the loss incurred by the approximations may be insignificant.

Figure 8 plots this expected number of comparisons (divided by 100000) for a full scan detecting all the occurrences in the text, as a function of the lengths of the original patterns. For example, the patterns of length 100 characters produced patterns of length 42 elements for 12 bit and 32 for 16 bit Tunstall codes. The values obtained for each length were averaged, which produced the dotted line curves in Figure 8. As a comparison, we also ran a regular search on the uncompressed text, producing the bold solid line. We see that for the longer patterns, the compressed search outperforms a decompress-then-search approach, by a factor of 5 for 12 bit, and of 5.3 for 16 bit Tunstall. The performance of the two Tunstall codes is very close, with a slight advantage of 16 bit for patterns longer than 50 characters.

7. CONCLUSION

We have introduced a new pattern matching problem, the SDMP, and suggested efficient ways to solve it. Both are based on suffix trees, but the first pre-processes the elements of the dictionary, whereas the second builds the trie for the given pattern. Which of the two should be preferred depends on the intended application and the given parameters. For an application to compressed matching, the generated solutions, in spite of their large number, are shown to perform much better than the original search in certain cases.

REFERENCES

[1] APOSTOLICO A., Fast gapped variants for Lempel-Ziv-Welch compression, *Information and Computation* **205** (2007) 1012–1026.

[2] BRATLEY P., CHOUEKA Y., Processing truncated terms in document retrieval systems, *Information Processing & Management* **18**(5) (1982) 257–266.

[3] BRISABOA N.R., FARIÑA A., LÓPEZ J.R., NAVARRO G., LÓPEZ E.R., A new searchable variable-to-variable compressor, *Proc. Data Compression Conference DCC–2010*, Snowbird, Utah (2010) 199–208.

[4] 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.

[5] Brisaboa N.R., Fariña A., Navarro G., Paramá J.R., Improving semistatic compression via phrase-based modeling, *Information Processing & Management* **47**(4), (2011) 545–559.

[6] Brisaboa N.R., Iglesias E.L., Navarro G., Paramá J.R., An efficient compression code for text databases, *Proc. European Conference on Information Retrieval ECIR'03*, Pisa, Italy, *LNCS* **2633**, Springer Verlag (2003) 468–481.

[7] Boyer R.S., Moore J.S., A fast string searching algorithm, Communications of the ACM **20** (1977) 762–772.

[8] Cantone D., Cristofaro S., Faro S.,, New Efficient Bit-Parallel Algorithms for the δ-Matching Problem with α-Bounded Gaps in Musical Sequences, *Proc. Prague Stringology Conference, PSC'08*, Prague, (2008) 170–184.

[9] Claude F., Navarro G., Self-Indexed Text Compression using Straight-Line Programs, *Proc. of the Symposium on Mathematical Foundations of Computer Science, MFCS* (2009) 235–246.

[10] Commentz-Walter B., A string matching algorithm fast on the average, *Proc. 6th International Coll. on Automata, Languages, and Programming ICALP'79*, Graz, Austria, *LNCS* **71**, Springer Verlag (1979) 118–132.

[11] Crochemore M., Iliopoulos C., Makris C., Rytter W., Tsakalidis A., Tsichlas K., Approximate string matching with gaps, *Nordic Journal of Computing* **9**(1) (2002) 54–65.

[12] Fredriksson K., Grabowski S., Efficient algorithms for pattern matching with general gaps, character classes and transposition invariance, *Information Retrieval* **11**(4) (2008) 335–357.

[13] Horspool R.N., Practical fast searching in strings, *Software – Practice & Experience* **10**(6) (1980) 501–506.

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

[15] Klein S.T., Efficient optimal recompression, *The Computer Journal* **40** (1997) 117–126.

[16] Klein S.T., Kopel Ben-Nissan M., On the Usefulness of Fibonacci Compression Codes, *The Computer Journal* **53**(6) (2010) 701–716.

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

[18] Klein S.T., Shapira D., On improving Tunstall codes, *Information Processing & Management* **47** (2011) in press, doi:10.1016/j.ipm.2011.01.005

[19] Lahoda J., Melichar B., Pattern matching in Huffman coded text, *Proc. Conference Information Society*, Vol **B**, Ljubljana, (2003) 274–279.

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

[21] 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.

[22] Navarro G., Raffinot M., Fast and Simple Character Classes and Bounded Gaps Pattern Matching, with Applications to Protein Searching, *Journal of Computational Biology* **10**(6) (2003) 903–923.

[23] Pinzon Y.J., Wang S., Simple algorithms for pattern matching with bounded gaps in genomic sequences, *Proc. Intern. Conf. on Numerical Analysis and Applied Math.*, (2005) 827–831.

[24] Sunday D.M., A Very Fast Substring Search Algorithm, *Communications of the ACM* **33**(8) (1990) 132–142.

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