Proceedings of the International Multiconference on Computer Science and Information Technology pp. 505–511
ISBN 978-83-60810-27-9
ISSN 1896-7094

Finding Patterns in Strings using Suffixarrays

Herman Stehouwer
Tilburg University
Tilburg Centre for Cognition and Communication
Tilburg, The Netherlands
Email: J.H.Stehouwer@uvt.nl

Menno van Zaanen
Tilburg University
Tilburg Centre for Cognition and Communication
Tilburg, The Netherlands
Email: M.M.vanZaanen@uvt.nl

Abstract—**Finding regularities in large data sets requires implementations of systems that are efficient in both time and space requirements. Here, we describe a newly developed system that exploits the internal structure of the enhanced suffixarray to find significant patterns in a large collection of sequences. The system searches exhaustively for all significantly compressing patterns where patterns may consist of symbols and skips or wildcards. We demonstrate a possible application of the system by detecting interesting patterns in a Dutch and an English corpus.**

I. Introduction

SYSTEMS that analyze large collections of sequential data, such as when searching for regularities in collections of texts, place strict requirements on the efficiency. Trivial implementations of such systems often yield correct results, but either take too long or use too much internal memory. These trivial implementations lead to limitations on the size of the data set that can be handled practically.

In this paper we propose a novel implementation of a system that can be used to search for regularities in sequential data. We show the practical applicability by searching for patterns in large text collections in two different languages. At the time of publication the source code of the implementation is publicly available on the web at http://ilk.uvt.nl/~stehouwer/.

To search the data sets efficiently, we initially used a data structure called a suffixtree. Suffixtrees are well-known data structures with many applications within the fields of bioinformatics, natural language processing, and many others. Ukkonen introduced an efficient online construction algorithm for suffixtrees in [9].

More recently, a similar data structure called suffixarray is often used instead of suffixtrees to search sequential data. A suffixarray is an ordered list of all suffixes in a sequence. This data structure was introduced in [6].

The reason for choosing suffixarrays instead of suffixtrees is the relatively large memory requirements of the suffixtrees. Gusfield discusses suffixtrees, their construction, and complexity requirements in great detail in his book [4]. The worst-case complexity of memory utilization of a suffixtree is $\Theta(m|\Sigma|)$ with m the length of the sequence and Σ the alphabet. This results in a suffixtree that can be searched in linear time (linear in the size of the search sequence). In contrast, suffixarrays are linear in their space utilization, regardless of the alphabet $|\Sigma|$.

In practical terms, a suffixtree build on the first 500.000 sentences from the British National Corpus took up 2.2GB. The suffixarray on the other hand only used 0.25 GB of main memory to be able to access the same data. From the last 10.000 sentences of the same corpus we generated a set of 70.000 queries. The suffixtree answered these queries in about 3 minutes, where the suffixarray system took around an hour.

Whereas usable suffixtree implementations can be found online[1], implementations of suffixarrays that we found available all have drawbacks. Often, the implementation limits the maximum size of the alphabet, which makes them unsuitable for our use.

In [1] several improvements to the original suffixarray data structure are combined to form what the authors call the enhanced suffixarray. One enhancement added to the enhanced suffixarray is the implementation of an encoding of an implicit tree structure. It is this implicit suffixtree structure that we will use in the application we describe here that allows us to find interesting patterns.

In this paper we will start by outlining the suffixarray data structure as well as its enhanced version. We concentrate on the implicit suffixtree structure that is made available through the enhancements. We will then show an application that finds patterns with skips (also called wildcards) in natural language data using suffixarrays by using the implicit suffixtree structure. Based on this application we show some experimental results on Dutch and English and in our conclusion we will suggest other possible applications of this kind of pattern finding.

II. Suffixarrays

In this section we will describe the techniques behind suffixarrays and their enhanced version.

A. Regular Suffixarrays

As introduced in [6] suffixarrays are relatively simple data structures that contain a lexicographically ordered list of all suffixes of the input sequence. The array does not contain explicit copies of the suffixes, but stores information on the suffixes implicitly in the form of an index on the original input. This means that much less memory is needed to store the suffixes compared to suffixtrees.

To build a suffixarray, we initialize an array of indexes describing all suffixes of the input sequence. This array is then

[1]See for instance http://ilk.uvt.nl/~menno/research/software/suffixtree.

sorted. When using a regular, efficient, sort algorithm, sorting takes $n \log n$ time. However, the ordering of the prefixes (of the suffixes stored in the array) can depend on multiple consecutive positions in the original input. This means that we have to perform sequence comparison which may need symbol comparisons of at most n positions. This results in a naive construction time of $n^2 \log n$ for a sequence of length n.

During the last several years, many sorting algorithms have been developed that can construct a suffixarray more efficiently in time requirements. The fastest algorithms run in $\Theta(n)$ time. These build a suffixtree first (which can be done in linear time) and then obtaining the sorted suffixes by a simple traversal of the suffixtree. Unfortunately, these algorithms need working space of at least $15 \times n$ [7].

In [7] a reasonably fast algorithm is proposed that needs working space of only $5 \times n$. This algorithm works by partially sorting the array into buckets which have the same x position prefix and afterwards sorting each of these buckets with a blind trie. This strategy is called *deep-shallow sort*.

An interesting aspect of suffixarrays is that suffixes that share a common prefix are grouped together in the array. This means that it can be used to locate the position and number of all infixes of an input sequence. This is done by finding all suffixes that start with the given infix. Since they are grouped together, they can be found efficiently.

B. Enhanced Suffixarrays

Several extensions to the regular suffixarrays have been proposed. The ones we will use (and have implemented) are described in [1]. The extensions combined with a regular suffixarray makes up an enhanced suffixarray. This data structure provides, among others, a different way of viewing the data, which is very similar to a suffixtree.

The enhancements store some information explicitly, which means that they require some additional storage in the shape of arrays. The first of those is the longest-common-prefix (or lcp) array. This array, parallel to the regular suffixarray, denotes the size of the prefix shared with the previous element. For instance, if the first element is *aard* and the second *aardvark* then the second element would have an lcp value of 4 as it shares a prefix of length four with the previous element in the suffixarray. This lcp array can be efficiently constructed in a single pass over the regular suffixarray.

The lcp values can be used to define the intervals, so called lcp intervals. An lcp interval can be seen as defining the interval corresponding to range of suffixes (in the suffixarray) with a specific prefix. An interval $[i \dots j], 0 \leq i < j \leq n$ with n the length of the sequence, is an lcp interval of the lcp value l if the following conditions hold (lcptab denotes the lcp array):

1) lcptab$[i] < l$,
2) lcptab$[k] \geq l$ for all k with $i + 1 \leq k \leq j$,
3) lcptab$[k] = k$ for at least one k with $i + 1 \leq k \leq j$,
4) lcptab$[j + 1] < l$.

TABLE I
AN ENHANCED SUFFIXARRAY ON THE SEQUENCE S = *acaaacatat* INCLUDING ITS LCPTAB AND CHILDTAB. THE FIELDS 1, 2 AND 3 OF THE CHILDTAB STAND FOR THE UP, DOWN AND NEXTINDEX FIELDS RESPECTIVELY. THIS EXAMPLE IS TAKEN FROM [1].

			childtab			
i	suftab$[i]$	lcptab$[i]$	1.	2.	3.	S[suffix]
0	2	0		2	6	*aaacatat$*
1	3	2				*aacatat$*
2	0	1	1	3	4	*acaaacatat$*
3	4	3				*acatat$*
4	6	1	3	5		*atat$*
5	8	2				*at$*
6	1	0	2	7	8	*caaacatat$*
7	5	2				*catat$*
8	7	0	7	9	10	*tat$*
9	9	1				*t$*
10	10	0	9			*$*

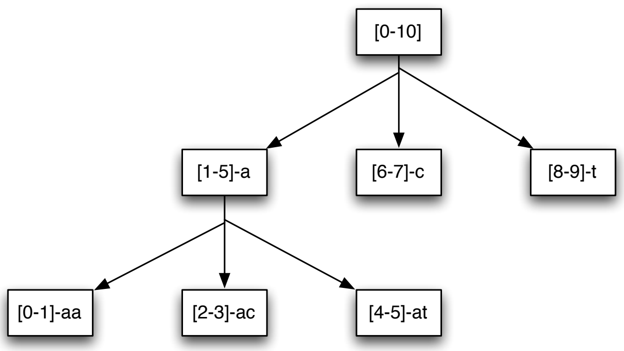

Fig. 1. An enhanced suffixarray on the sequence S = *acaaacatat* produces the lcp interval tree shown. This example is taken from [1].

The lcp intervals can have smaller lcp intervals embedded within them recursively. These recursive intervals can be seen as a tree structure and is called an lcp interval tree. This lcp interval tree is implicit. Furthermore, it has the same structure as the suffixtree if it were built based on the sequence in the suffixarray.

We would like to be able to access the implicit suffixtree structure in an efficient way. In order to do this we store the jumps through the suffixarray that we need for top-down traversal of the implicit suffixtree in an extra support array. This extra support array is called the child table in [1]. With this extra information we can determine the longest-common-prefix and its child intervals for each interval with a simple array lookup. For this to work we need to start with a valid interval. Luckily the interval $[0 \dots n]$ is always valid.

The construction of the child table can be done, just like the lcp value array, in a single pass over the suffixarray. However, building the child table depends on the presence of the lcp table, so after constructing and sorting the regular suffixarray we have to perform two more passes over the suffixarray to fill the support structures that allow access to the implicit suffixtree structure.

An example of a regular suffixarray enhanced with the lcp values and the childtab can be found in Table I. We also show the corresponding lcp interval tree in Figure 1.

III. Related Programs

In order to understand the context of the suffixarray-based patternfind program we will briefly describe related systems that are available.

The Ngram Statistics Package [2] can perform statistical tests on n-grams sampled from a window of size k. Effectively, this means that the package can look for sparse n-grams. It supports many statistical tests such as the Dice Coefficient, Fishers test, and Mutual Information. The Ngram Statistics Package is available on Pedersen's website at http://www.d.umn.edu/~tpederse/nsp.html.

Daciuk has developed several Finite-State based libraries and tools, including one for compressing FSA-based language models [3]. Such compression is related to the approach we take to find patterns with our patternfind software. This software is available at http://www.eti.pg.gda.pl/katedry/kiw/pracownicy/Jan.Daciuk/personal/fsa.html.

There are also software packages available for extracting significant n-grams and for performing statistical subsequence reduction as described in [8]. One implementation is that of Zhang, which is available at http://homepages.inf.ed.ac.uk/lzhang10/ngram.html.

PAFI is a piece of software for finding frequents patterns in large and diverse databases. In [5] some of the (graph-based) techniques that underly the system are described. The program is available from http://glaros.dtc.umn.edu/gkhome/pafi/overview together with related software.

IV. Implementation

Our implementation of the suffixarray is done in template-based C++. This leads to an extremely flexible implementation while making only limited demands on the container type. The template types used in the suffixarray must support only the following basic functionality:

1) The subtype has to support the following operators:
   a) The comparison operator $<$,
   b) The comparison operator $>$,
   c) The comparison operator $!=$,
   d) The comparison operator $==$,
   e) The assignment operator $=$,
   f) The increment operator: $++$,
   g) The output operator: $<<$.
2) The type contents must be accessible via several ways:
   a) Via the $[x]$ construct,
   b) Via the iterators x.begin() and x.end(),
   c) Via the const_iterators x.begin() and x.end().

Furthermore, when building a suffixarray, the input sequence must end with a unique element that is largest when compared to any other element in the sequence according to the comparison operators. This additional requirement allows us to eliminate several bounds checks in the heart of the sorting code.

In our implementation of the suffixarray algorithm, space utilization for a data collection of length n is

$$n \times \text{sizeof(index)} + 4 \times n * \text{sizeof(symbol)} + \text{exceptions}$$

This includes the additional arrays that are used to implement the enhancements as described in Section II-B.

In practice, building the suffixarray using our implementation is very time efficient (approximately 2–3 minutes for 1 million natural language sentences). However, for input sequences that have a high longest-common-prefix (lcp), our implementation will not be particularly efficient, due to the nature of the sorting stage. We use a deep-shallow sorting strategy with a blind trie, which was introduced by Manzini and Ferragina in [7]. Natural language data, however, which is our area of interest, is sorted very efficiently as it is by nature low in lcp.

We have implemented the enhancements that are needed for the implicit suffixtree structure as described in [1]. The hints and algorithms described in the article form the basis of the practical implementation.

V. Programs

The package contains three user programs. The usage of these programs will be described in more detail in the next three sections.

A. *Patternfind*

The pattern finding program looks for frequently occurring skip-grams as patterns. A skip-gram is similar to a regular n-gram with the possibility of containing non-consecutive skip positions. A skip represents a wildcard, which may match any number of arbitrary symbols. In article we will refer to such skip positions in patterns as **SKIP**. In Figure 2 a simple skip-gram is shown visually in a simple tree-structure.

1) Implementation: The pattern finding is done as an exhaustive, breadth-first search over the implicit suffixtree with pruning. During the search, the internal representation keeps track of the pattern represented so far, for instance *A *SKIP*A*, together with a set of [lcp-interval, depth] pairs.

An [lcp-interval, depth] pair represents a path through the implicit suffixtree. The lcp interval has an associated lcp value, which may be larger than the number of steps that have been taken to reach the lcp interval in the specific path. For instance, if the sequence S = *abcabc* is turned into a suffixarray, the lcp interval $[0, 1]$ has an lcp value of three as the first three characters of the two suffixes starting with *a* will be *abc*. Because we would like to be able to represent the step *a* we also associate a depth with the lcp interval to indicate the exact position in the implicit suffixtree. This example is illustrated in Table II.

The suffixarray is searched using the patterns, where for each pattern, all single steps are taken. A single step can be an explicit symbol as defined by the pattern or a skip position if the previously taken step was not a skip. Only continuations of the (sub-)pattern that are directly reachable from the set of [lcp-interval, depth] pairs are considered.

TABLE II
ILLUSTRATING THE SUFFIXARRAY, INCLUDING THE LCP TABLE, OF THE SIMPLE SEQUENCE *abcabc*.

i	suftab[i]	lcptab[i]	suffix
0	3	0	*abcabc$*
1	0	3	*abc$*
2	4	0	*bcabc$*
3	1	2	*bc$*
4	5	0	*c$*
5	2	1	*cabc$*
6	6	0	*$*

Obviously applying this approach directly results in an exponential explosion in the number of different patterns that are found in a dataset. To keep this amount manageable a threshold can be set. Based on this threshold, only interesting patterns are retained.

For each pattern discovered, a prune value is computed. This value depends on the compressibility score, which consists of the number of items in the pattern minus one, not counting the skip positions, multiplied by the number of times it occurs in the sequence. In this case, the compression rate of a unigram pattern is zero. To remove this limitation, we use the frequency in the sequence for patterns of length one as the compressibility score.

Based on the computed prune value, we decide whether the pattern should be pruned or not. If the score of the new pattern exceeds the prune value it is added to the pattern list for the next round.

After taking a few of such steps a pattern emerges. In Figure 2 we show a very simple implicit suffixtree and highlight the effect of the pattern *A *SKIP* A* on that example. All dark-grey nodes in the example represent [lcp-interval, depth] in the suffixarray that would occur in the cloud of the pattern in this example.

After having considered all valid steps during the pattern search phase, the remaining patterns (which have a score above the pruning threshold) are written to output. Only the patterns above a certain threshold are given to the user. The threshold can be set separately from the prune value.

The program itself is implemented in template-based C++ and in its distributed form can be applied to a sequence of text-based word tokens which are read from a file. Internally, these are mapped to a list of numbers. The suffixarray is built on the sequence of numbers. This list of numbers can be transformed back into a sequence of words using the one-to-one mapping, which is what happens during the output of the patterns. This transparent, internal mapping is done, because the comparison operators on integers are significantly faster than those on sequence objects. Additionally, a list of integers occupies less memory[2].

[2]Personally, we run a custom version of the code that takes as input a pre-mapped corpus that we have to manually map back and forth. By doing this we do not have to load the mapping in memory as well. For usability reasons we automated this process in the released tool.

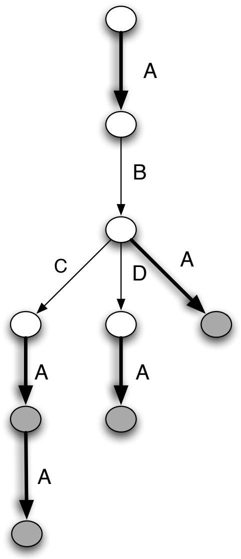

Fig. 2. An example of the pattern *A *SKIP* A* found in a simple tree. Notice how the **SKIP** can have different lengths. All dark-grey nodes represent members of the [lcp-interval, depth] set associated with the pattern *A *SKIP* A*.

TABLE III
THE OUTPUT OF THE PATTERNFIND PROGRAM WHEN RUN WITH THE −HELP OPTION.

```
                         patternfind -h
Usage: ./patternfind[OPTION]...
This program reads in a corpus and stores it in a suffixarray.
It then searches for all significant patterns in the corpus.
  -h, --help        Show this help and exit
  -f, --file FILE   Filename of the corpus to be read
  -p, --prune PRUNE-VALUE    The value at which to start pruning the found
          patterns.
        Determined by the number of occurrences and the size of the pattern.
  -o, --output PRINT-VALUE     The value at which to start printing the found
          patterns,  works the same as the prune value.
        If it is smaller or equal to the prune value it has no extra effect.
  -s --smallestskip SKIP    Minimum number of positions to skip for a SKIP
          part of a pattern.
  -l --largestskip SKIP    Maximum number of positions to skip for a SKIP
          part of a pattern.
```

2) Usage: The program that searches for compressing skip-grams contained in the suffixarray is called patternfind. As described above, this program identifies patterns that occur frequently and hence lead to a compression higher than a threshold. When patternfind is run with the `-help` option it gives the output as shown in Table III.

TABLE IV
THE OUTPUT OF THE PATTERNPOSITIONS PROGRAM WHEN RUN WITH THE −HELP OPTION.

```
                     patternpositions -h
Usage: ./patternpositions[OPTION]...
This program reads in a corpus and stores it in a suffixarray.
        It then reads in a file of patterns and for each of those patterns
        returns all occurring positions in the corpus.
  -h, --help        Show this help and exit
  -f, --file FILE   Filename of the corpus to be read
  -s --smallestskip SKIP    Minimum number of positions to skip for a SKIP
          part of a pattern. Should be the same value as used in patternfind.
  -l --largestskip SKIP    Maximum number of positions to skip for a SKIP
          part of a pattern. Should be the same value as used in patternfind.
  -p, --patterns FILE    The file containing the patterns to output the
          positions of.
         As output by patternfind. I.e. patternfind options > patternfile;
        patternpositions options -patterns patternfile
```

The most important parameters specify an input sequence (in a file) and a prune value. The other values are optional and only modify the default behavior of the program. For example, we can look at the patterns found in the included README file by executing the following command. We will set the prune value to 10 and limit the skip to skips of the size 1–3.

```
./patternfind -f ../README -p 10 -s 1 -l 3
```

All status messages are written to standard error and all patterns are written to standard output. A pattern is presented together with a number representing its compressibility score on which pruning was done. **SKIP** denotes a skip position. At that point the pattern skips a flexible space of, in this case [1–3] positions.

B. Patternpositions

The patternpositions program searches a sequence for occurrences of patterns such as those output by the patternfind program. To specify the exact behavior of the skips, it takes the same options as the patternfind program. It takes one extra argument, namely the patterns file from which all the patterns will be read. The output of its help function can be seen in Table IV.

For each pattern in the pattern file, patternposition will print all the positions of occurrences of that pattern in the corpus. We can for instance run the program on the patterns we found in the README file with patternfind in the following way:

```
./patternpositions -f ../README -s 1 -l 3\
  -p ../README.patterns
```

Just like with the patternfind program we print all results to the standard output and all the status messages to the standard error.

C. Suffixarray

The final program included in the distribution is called suffixarray. This program has only one option, the file containing the sequence which should be used to build a suffixarray. This file should be a plain text file containing a white space separated sequence of tokens, such as words.

Running the program will result in a suffixarray being build from the specified sequence after which the program waits for input. At this point the program will answer simple n-gram queries by reporting the number of times the typed n-gram occurs in the sequence contained in the suffixarray. In these n-grams it is allowed to use simple single-positions wildcards, represented by the token *.

VI. Experimental Results

To show practical use, we performed preliminary pattern finding experiments on two natural language corpora: an English-language corpus, the British National Corpus (BNC) and a Dutch-language corpus, our own collection of texts of local newspapers known internally as the ILK-BDEDGE corpus.

The British National Corpus is a corpus of around 100 million words of both spoken and written English. We use

TABLE V
The 40 highest ranking patterns found in one million sentences of the British National Corpus. The score of the pattern is shown as well.

```
460718 the *SKIP* of
308872 of the
252720 the *SKIP* .
240072 the *SKIP* ,
235770 , *SKIP* ,
208494 , *SKIP* the
201894 . The
195756 the *SKIP* of the
195478 in the
191296 the *SKIP* the
176920 , and
173836 . "
162396 of *SKIP* ,
158076 . *SKIP* the
156170 of *SKIP* .
154302 . *SKIP* ,
146010 a *SKIP* of
138120 the *SKIP* and
127728 to *SKIP* the
122818 , the
111534 to the
108086 to *SKIP* .
107156 and *SKIP* .
104474 , *SKIP* and
98664 in *SKIP* ,
97588 in *SKIP* .
96626 of *SKIP* and
96318 and *SKIP* ,
96063 the *SKIP* of *SKIP* .
94191 the *SKIP* of *SKIP* ,
93480 " *SKIP* "
90878 . *SKIP* is
88758 a *SKIP* ,
88290 and *SKIP* the
88234 a *SKIP* .
85738 , *SKIP* a
83964 . *SKIP* was
81498 the *SKIP* to
81458 to *SKIP* ,
80346 , *SKIP* of
```

TABLE VI
10 patterns found in one million sentences of the British National Corpus. These patterns are in the middle of the score-ranking. The score of the pattern is shown as well.

```
1872 with the *SKIP* a
1872 was *SKIP* in a
1872 to be *SKIP* , and
1872 the *SKIP* between *SKIP* and *SKIP* .
1872 that it *SKIP* not
1872 over the *SKIP* the
1872 of all the
1872 of *SKIP* in the *SKIP* the
1872 in the *SKIP* of *SKIP* "
1872 in *SKIP* will
```

TABLE VII
10 patterns found in one million sentences of the British National Corpus. These patterns are the last of the score-ranking. The score of the pattern is shown as well.

```
1002 . *SKIP* the second
1002 . *SKIP* the *SKIP* would
1002 . *SKIP* most of
1002 . *SKIP* help
1002 . " *SKIP* , *SKIP* he *SKIP* . "
1002 , *SKIP* to *SKIP* them
1002 , *SKIP* the *SKIP* this
1002 , *SKIP* fact that
1002 , " *SKIP* Mr
1001 . " *SKIP* , " he said ,
```

TABLE VIII
THE 40 HIGHEST RANKING PATTERNS FOUND IN ONE MILLION SENTENCES OF THE CORPUS OF REGIONAL DUTCH NEWSPAPERS. THE SCORE OF THE PATTERN IS SHOWN AS WELL.

```
286710  .       "
263612  van     de
238974  de      *SKIP*    .
237896  de      *SKIP*    van
207394  .       De
171850  de      *SKIP*    de
166406  .       *SKIP*    de
147504  in      de
132682  van     *SKIP*    .
128690  te      *SKIP*    .
120294  de      *SKIP*    van     de
119124  ,       *SKIP*    de
113170  .       *SKIP*    is
112282  het     *SKIP*    van
108446  de      *SKIP*    ,
103996  in      *SKIP*    .
102382  het     *SKIP*    .
95998   een     *SKIP*    .
94398   .       Het
92986   de      *SKIP*    in
92720   van     het
80518   .       *SKIP*    van
79314   .       *SKIP*    het
77822   en      *SKIP*    .
77746   (       *SKIP*    )
77211   van     de      *SKIP*    .
76828   het     *SKIP*    de
74458   "       ,
74276   ’       *SKIP*    ’
72732   op      de
72714   van     *SKIP*    ,
72386   een     *SKIP*    van
72266   de      *SKIP*    het
72176   in      het
71564   ,       *SKIP*    ,
68942   de      *SKIP*    en
67518   ,       *SKIP*    en
67102   voor    de
66948   aan     de
63842   ,       *SKIP*    .
```

TABLE IX
10 PATTERNS FOUND IN ONE MILLION SENTENCES OF THE CORPUS OF REGIONAL DUTCH NEWSPAPERS. THESE PATTERNS ARE IN THE MIDDLE OF THE SCORE-RANKING. THE SCORE OF THE PATTERN IS SHOWN AS WELL.

```
1515    op  het *SKIP*    een
1515    nog *SKIP*    van de
1515    naar    *SKIP*    .   Het
1515    is  het een
1515    is  *SKIP*    de  *SKIP*    van de  *SKIP*    .
1515    het *SKIP*    in  *SKIP*    een
1515    een *SKIP*    .   *SKIP*    :
1515    door    de  gemeente
1515    de  *SKIP*    .   *SKIP*    uit
1515    bestuur van de
```

TABLE X
10 PATTERNS FOUND IN ONE MILLION SENTENCES OF THE CORPUS OF REGIONAL DUTCH NEWSPAPERS. THESE PATTERNS ARE THE LAST OF THE SCORE-RANKING. THE SCORE OF THE PATTERN IS SHOWN AS WELL.

```
1002  .  *SKIP*  dat  *SKIP*  er
1002  .  *SKIP*  daarom
1002  .  *SKIP*  acht
1002  .  *SKIP*  Volgens  *SKIP*  is
1002  .  *SKIP*  J.  van
1002  ,  *SKIP*  of  *SKIP*  ,
1002  ’  ,  ’  *SKIP*  ’  *SKIP*  ’  *SKIP*  ’
1002  ’  *SKIP*  of  *SKIP*  ’
1002  "  Wij  zijn
1001  ,  2  .  *SKIP*  ,  3  .  *SKIP*  ,
```

a 1 million sentence chunk for our pattern finding experiment. This chunk consist of around 21 million words.

The ILK-BDEDGE is a corpus consisting of material from the Brabants Dagblad, Eindhovens Dagblad and De Gelderlander. These are all regional, Dutch, newspapers. Again, we took a chunk of 1 million sentences from the corpus to perform experiments on. This chunk consists of around 16 million words. The average sentence length is somewhat smaller than that of the BNC.

On each of the 1 million sentences we ran the pattern finder program with a prune value of 1000. The 40 highest ranked patterns found on the BNC are shown in Table V. Likewise for the corpus of Dutch local newspapers, we show the 40 highest ranked patterns in Table VIII. For English we find 23,705 patterns and for Dutch 18,004 patterns in total that make the threshold.

The program took slightly less than 32 hours to run on the ILK-BDEDGE corpus part we used. During that time its memory usage peaked at approximately 700MB.

We see in these tables that function words such as determiners, that delimit the overall structure of the sentences, are found the most. Unsurprisingly, all these words together with punctuation marks are amongst the most frequent tokens in both languages. For instance in English we find as the most compressing pattern *the *SKIP* of* and in the Dutch list the corresponding pattern *de *SKIP* van* also scores very high.

When we look at the patterns found in the middle of the pack we see less generic and possibly more useful patterns, such as *with the *SKIP* a*. These patterns from the middle can be found in Table VI for English and Table IX for Dutch.

The lowest-scoring patterns are shown in Table VII for English and Table X for Dutch. These patterns start with punctuation and after that specialize in a specific way. Patterns with very low, compared to the size of the corpus, scores are not generally very informative. These patterns could be removed automatically by increasing the threshold.

VII. CONCLUSION

In this paper, we have described a package that contains an efficient, flexible and practical implementation of the suffixarray data-structure. Three programs are provided, allowing for efficient search in a large sequence of symbols and for finding interesting, compressing patterns.

The program that searches for compressing patterns has been applied to a collection of natural language texts. The patterns found describe the global structure of the sentences in which they occur. These patterns can be used by linguists to help them find naturally occurring constructions in language.

With respect to future work, this system could be applied in different areas. For instance, another application of the patternfind program can be found in the area of morphology. Instead of providing a collection of sentences, as shown here, a collection of words is provided to the system and patterns that occur regularly within words can be identified. This may lead to automatically found descriptions of the morphology of words.

Another possible application would be to apply the found patterns of several languages to a parallel corpus. The patterns can be used to identify often occurring patterns in the different languages, which can then be used to align translations of texts. Also, in the same line, the aligned patterns can be used to enrich a phrase-based machine-translation system.

Additionally, we would like to write several extensions to the programs, the simplest of them being a small program that identifies contexts for a given pattern in a specific position. This information is directly accessible in the suffixarray data structure, but cannot easily be identified with the current programs. Another, more complex addition we would like to write, is a program that given a pattern finds the most pertinent content that matches the skips in that pattern.

Finally, we would like to provide a version that is annotation-aware, allowing it to find patterns not only with skips that match any symbol, but taking specific annotation layer items into account. This could lead to patterns such as *the N of the N*, where the *N* specifies a noun as described by an annotation layer. This final improvement should also be extended to the program that tries to find the pertinent content that fills up skips in particular patterns.

References

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] S. Banerjee and T. Pedersen. The design, implementation, and use of the Ngram Statistic Package. In *Proceedings of the Fourth International Conference on Intelligent Text Processing and Computational Linguistics*, pages 370–381, Mexico City, February 2003.

[3] J. Daciuk and G. van Noord. Finite automata for compact representation of language models in nlp. In *CIAA '01: Revised Papers from the 6th International Conference on Implementation and Application of Automata*, pages 65–73, London, UK, 2002. Springer-Verlag.

[4] D. Gusfield. *Algorithms on Strings, Trees and Sequences*. University of Cambridge, Cambridge, 1997.

[5] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. Technical report, IEEE Transactions on Knowledge and Data Engineering, 2002.

[6] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[7] G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.

[8] M. Nagao and S. Mori. A new method of n-gram statistics for large number of n and automatic extraction of words and phrases from large text data of japanese. In *In COLING-94*, pages 611–615, 1994.

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