Parallel Suffix Arrays for Linguistic Pattern Search

Johannes Goller
Macmillan, Digital Science
Chiyoda Bldg., 2-37 Ichigayatamachi
Shinjuku-ku, Tokyo
jogojapan@gmail.com

Abstract

The paper presents the results of an analysis of the merits and problems of using suffix arrays as an index data structure for annotated natural-language corpora. It shows how multiple suffix arrays can be combined to represent layers of annotation, and how this enables matches for complex linguistic patterns to be identified in the corpus quickly and, for a large subclass of patterns, with greater theoretical efficiency than alternative approaches. The results reported include construction times and retrieval times for an annotated corpus of 1.9 billion characters in length, and a range of example patterns of varying complexity.

1 Introduction

Empirical linguistic studies require access to large corpora of text, and they benefit greatly when the text is stored in a form that enables the efficient retrieval of specific elements, such as sentences that match a pattern defined by a linguist. The size and contents of the corpus, the type and structure of its annotations, and the form of patterns involved vary greatly; the present paper deals with the requirements of only a subset of linguistic studies, which are characterized as follows:

- The corpus is large (hundreds of millions of words), but not extremely large (hundreds of millions of documents);
- Annotations exist in any number of layers, for example a layer of part-of-speech (POS) annotations and a layer of semantic role labels, but the annotations on each individual layer are non-overlapping and non-ambiguous;
- A pattern is essentially a regular expression, made up of literals (to be matched against the text), annotations (each with a specification of the layer it is expected to be found in) and wildcard elements ("gaps");
- The retrieval results are expected to be delivered within seconds or minutes (that is, not necessarily as fast as web search), and to be comprehensive (that is, to contain all matches, not only the top-N defined by some relevancy ranking);
- New patterns are generated constantly, perhaps by many different users or automated programs in parallel, while the text is largely static.

Corpus search engines that respond to a similar, albeit not identical, set of requirements include the Corpus Workbench[1], WebCorp Linguist's Search Engine[2] and Manatee/Bonito (Rychlý, 2007). The implementation of all of these systems relies on the principle of inverted files, which is the main alternative to the suffix arrays presented here[3]. Both approaches are described and briefly compared in section 2, a direct comparison is also available in (Puglisi et al., 2006). Sections 3 and 4 introduce the concept of parallel suffix arrays and describe how it enables annotations and complex pattern search, including patterns equivalent to finite state machines. Section 5 describes results obtained using an actual implementation of parallel suffix arrays.

2 The Two Main Approaches to Indexing

2.1 Inverted Files

The concept of inverted files requires the text to be *tokenized*, that is, to be segmented into tokens

[1] http://cwb.sourceforge.net/
[2] http://www.webcorp.org.uk/
[3] Suffix arrays are frequently used for n-gram analyses (e.g. Yamamoto and Church (1998)), but without the ability to process complex search patterns.

Proceedings of Recent Advances in Natural Language Processing, pages 487–494,
Hissar, Bulgaria, 12-14 September 2011.

(usually roughly equivalent to words). The index consists of a searchable dictionary of the tokens (e.g. a hash table or sorted list), and a link connecting each token with its inverted list, i.e. the list of positions where the token is found (where the position of a token is defined as the token offset, i.e. the number of tokens to its left).

The match result for a search pattern that consists of a single token t is then readily retrieved by determining the dictionary entry corresponding to t (which if hashing is used typically takes $O(|t|)$ time, where $|t|$ is the length of t in characters) and returning the entire inverted list I_t. The length of the list corresponds to the number of occurrences of t, $\text{occ}(t)$. If the pattern is a sequence of tokens $P := t_1, \ldots, t_r$, the retrieval strategy is to determine all inverted lists in $O(|t_1|) + \ldots + |t_r|)$ time, to then identify the inverted list of the least frequent token, i.e. I_{t_μ} such that $\mu = \text{argmin}_i \text{ occ}(t_i)$, and to finally check for each of the positions $p \in I_{t_\mu}$ whether it lies in a match for the entire pattern P. That requires, for each p a look-up in the remaining $r-1$ inverted lists, specifically, for each $1 \leq k < \mu$, a look-up to check whether $p - k \in I_{t_{\mu-k}}$, and for each $1 \leq k < (r - \mu)$ to check whether $p + k \in I_{t_{\mu+k}}$. Since inverted lists are usually stored as sorted lists of integers, a look-up in I_{t_i} requires $O(\log \text{ occ}(t_i))$ time, hence the total time taken to identify all matches for P is

$$O\big(\sum_k |t_i| + \text{ occ}(t_\mu) \sum_{k \neq \mu} \log \text{ occ}(t_k)\big) \qquad (1)$$

Storing annotations in the index is straightforward: Modify the inverted lists so as to store positions as character offsets (rather than token offsets), and the length of t in characters along with each occurrence of t. Annotations can then be indexed in the same way as ordinary tokens, with character offset and length, and the procedure above can be modified so as to take into account the length of each t_i when computing the positions of adjacent tokens. This enables patterns using a mix of text and annotations, i.e. with some of the t_i referring to text, others to annotation. The time bound of (1) is unchanged.

2.2 Suffix Arrays

A *suffix array* is any representation of the lexicographically sorted list of all suffixes of a text, where *suffix* is defined as any substring beginning somewhere in the text and ending at the end of the text, i.e. there are n suffixes in a text of length n. Rather than storing copies of all the substrings, the suffix array is usually represented as a list of n integers, each indicating the starting position of a suffix. An example of this is shown in Fig. 1: The suffix array itself consists only of the integer list SA; the lower part of Fig. 1 shows the strings corresponding to each position, written vertically. Suffix arrays have an important property related to substring searches: Given a text T, its suffix array SA and a search pattern P, the set of starting positions of matches for P in T forms a continuous range in SA, as each match is the initial part of a suffix of T. Because of the lexicographical sorting, these suffixes must be adjacent to each other in the suffix array. For example, the set of matches for substring `ab` in Fig. 1 is the range $[2;3]$ of SA (corresponding to positions 4 and 1 of T). This shall be called the **range property** of suffix arrays.

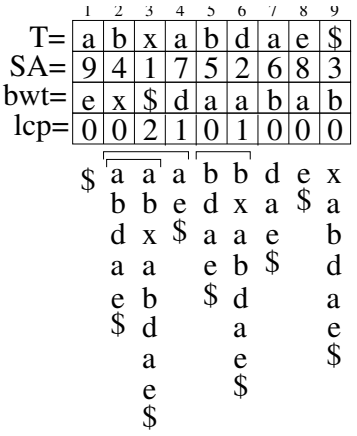

Figure 1: Suffix array SA for the string $T =$`abxabdae$`, along with auxiliary data structures *bwt*, *lcp* and "brackets" indicating the match ranges for substrings `ab`, `a` and `b`.

Recent improvements in search algorithms for suffix arrays, cf. Navarro and Mäkinen (2007), make it possible to identify the match range for P in $O(|P|)$ time[4], and since no tokenization is required, recombining matches for individual tokens as in the case of inverted files is unnecessary. However, it is impossible to store annotation-related information in the suffix array. The follow-

[4]Strictly speaking, the time is bound by $O(|P|(1 + \log|\Sigma|/\log\log n))$, where $|\Sigma|$ is the size of the alphabet. However that is asymptotically equivalent to $O(|P|)$ when the alphabet is as much smaller than the text as it is the case for large-scale natural-language corpus search. See Navarro and Mäkinen (2007, 42) for details.

ing two sections describe a new concept, *parallel suffix arrays*, and how it enables annotations and more powerful search patterns.

3 Parallel Suffix Arrays

The first step is to allow annotations to enter the index. In the following it is assumed that a text $T \in \Sigma^*$ of length n is given, and one layer of q annotations

$$A = ((a_1, p_1, \ell_1), (a_2, p_2, \ell_2), \ldots, (a_q, p_q, \ell_q))$$

such that each annotation (a_i, p_i, ℓ_i) consists of a label $a_i \in \Sigma^*$, a starting position $p_i < n$ and a length ℓ_i. p_i indicates where in T the substring annotated with a_i starts, ℓ_i indicates the number of T-characters it covers. For example, given

$$T = \text{is but a dream within a dream}$$

and POS-annotations V, Conj etc., the annotation layer might look like this:

$$A = ((\text{V}, 1, 2), (\text{Conj}, 4, 3), (\text{Det}, 8, 1), (\text{N}, 10, 5), (\text{Prep}, 16, 6), (\text{Det}, 23, 1), (\text{N}, 25, 5)) .$$

There are two ways to bring these annotations into the suffix-array-based index for T:

Method 1: Single-integer annotations. Three steps need to be performed: (1) Each distinct annotation label is mapped to a unique integer (e.g. using a hash table), that is, a new annotation alphabet Λ is created, in which each annotation is represented as one integer. (2) An extra integer is introduced in Λ, below represented by $\emptyset$, which is used as a dummy annotation for all areas of T that are not covered by any element of A (in the example above, this applies to the space characters between words). (3) A is replaced by a string $A' \in \Lambda^*$ containing the new annotation symbols in the order of the T-positions they refer to, and a bitvector $B^{T \leftrightarrow A}$ of length n indicating the starting positions of annotations relative to T. The example above now becomes:

$$A' = 1\emptyset 2\emptyset 3\emptyset 4\emptyset 5\emptyset 3\emptyset 4$$
$$B^{T \leftrightarrow A} = 10110011110000110000011110000 ,$$

where V has been mapped to 1, Conj to 2, and so forth. The next step is to construct a suffix array $\text{SA}_{A'}$ from the Λ-string A', along with auxiliary data structures required for fast searches, cf. Navarro and Mäkinen (2007). That enables fast searches for sequences consisting solely of annotations. It will later be shown how the bitvector is used to accomplish searches for mixed patterns, that is, patterns that contain both, T-sequences and A-sequences.

Method 2: Complex annotations. In some situations annotations are themselves complex and one would like to be able to search inside them, rather than mapping them to atomic integers. This is accomplished by appending a new character $\sharp \notin \Sigma$ to every label a_i as a separation mark, and then concatenating all labels to a new string A':

$$A' = \text{V}\sharp\text{Conj}\sharp\text{Det}\sharp\text{N}\sharp\text{Prep}\sharp\text{Det}\sharp\text{N}\sharp$$

In addition, two bitvectors $B^{T \leftrightarrow A}$ and $B^{A \leftrightarrow A}$ are defined, the former in the same way as in method 1, while the latter is of length $|A'|$ and has a 1 wherever a new annotation starts in A':

$$B^{A \leftrightarrow A} = 10100001000101000010010$$

Again, a suffix array $SA_{A'}$ for A' enables searching for substrings of annotations as well as sequences of annotations. The $\sharp$-symbols prevent undesired matches across annotation-boundaries.

How the bitvectors are used for mixed T/A' patterns. Both the bitvector of the first, and the bitvectors of the second method need to undergo an indexing process, during which a *rank* index and a *select* index are generated for each bitvector, defined as follows: Let B be a bitvector of length b and $i, j < b$, then

$$\texttt{rank}_B(i) := \text{the total number of 1s in } B[1..i]$$
$$\texttt{select}_B(j) := i \text{ s.t. there are } j \text{ 1s in } B[1..i] .$$

Using techniques described by Jacobson (1989), it is possible to construct, in $O(b)$ time, data structures that implement these functions, such that a lookup can be performed in $O(1)$ time and no more than $b + o(b)$ bits of space are consumed in total (including the bitvector itself). In the case of single-integer annotations (method 1), $\texttt{rank}_{B^{T \leftrightarrow A}}$ and $\texttt{select}_{B^{T \leftrightarrow A}}$ are constructed; in the case of complex annotations, these and $\texttt{rank}_{B^{A \leftrightarrow A}}$ and $\texttt{select}_{B^{A \leftrightarrow A}}$ are constructed. In addition, in both cases the inverse suffix arrays for T and A' must be computed and stored in memory: Given a suffix array SA, its inverse is defined as

$$\text{invSA}[j] := i \text{ such that } \text{SA}[i] = j ,$$

and invSA can be generated from SA in linear time. To see how these data structures work together, consider a mixed pattern $\sigma\lambda$, where $\sigma \in \Sigma^*$ is a substring match against T and λ is a substring match against the annotations. We first assume that method 1 was used, hence that $\lambda \in \Lambda^*$ is a sequence of annotations mapped to integers. The next step is to search the suffix arrays and determine the match ranges (l_σ, r_σ) for σ in SA_T and (l_λ, r_λ) for λ in $\mathrm{SA}_{A'}$. Clearly, the number of occurrences of σ in T is $\mathrm{occ}(\sigma) = r_\sigma - l_\sigma$, the number of matches for λ is $\mathrm{occ}(\lambda) = r_\lambda - l_\lambda$. We must now check, for each σ-match, whether it is followed by a λ-match. Let $l_\sigma \le x < r_\sigma$ one of the σ-matches. It begins at position $p = \mathrm{SA}_T[x]$ of T and it is $|\sigma|$ characters in length. Hence it is followed by a λ-match if and only if an A-annotation starts at $p + |\sigma|$ and that annotation corresponds to a λ-match in A', which is the case iff the corresponding position in A' is a suffix in the match range (l_λ, r_λ). We therefore verify, for the candidate offset $q := p + |\sigma|$:

$$A\text{-element exists:} \quad B^{T\leftrightarrow A}[q] = 1 \qquad (2)$$

$$\text{Location in } A'\text{:} \quad q' := \mathtt{rank}_{B^{T\leftrightarrow A}}(q) \qquad (3)$$

$$\text{Is } q' \text{ a } \lambda\text{-match:} \quad l_\lambda \le \mathrm{invSA}_A[q'] < r_\lambda \qquad (4)$$

If SA and invSA are available for random access, all of the above can be tested in $O(1)$ time, hence it takes $O(\mathrm{occ}(\sigma))$ time to compute the set of $\sigma\lambda$-matches from the two individual match ranges. Moreover, the procedure works in the reverse direction, too, starting from the λ-matches and determining those among them that are preceded by a σ-match (using `select` instead of `rank`; the time consumption becomes $O(\mathrm{occ}(\lambda))$). Hence it is possible to choose the matching direction according to whichever part of the pattern has fewer matches, i.e. let $\mathrm{occ}_\mu := \min(\mathrm{occ}(\sigma), \mathrm{occ}(\lambda))$, then the match combination can be computed in $O(\mathrm{occ}_\mu)$ time.

Without giving a detailed proof, we note that this result can be extended to general sequential patterns $t_1 \cdots t_r$, $t_i \in \Sigma^*, \Lambda^*$: The match combination time depends only on the least frequent (i.e. most specific) element t_μ, that is, including the time taken to determine the match range for each t_i, the total asymptotic time is

$$O\Big(\sum_k |t_k| + \mathrm{occ}(t_\mu)\Big)\,, \qquad (5)$$

which is obviously better than with inverted files, where the match combination time depends on the frequency of all elements, as shown in (1). This shall be called the **least-frequency property** of parallel suffix arrays[5]. It should also be noted that for subsequences $t_e \cdots t_f$ such that all elements refer to the same layer, i.e. $\forall t_i \in \Sigma^*$ or $\forall t_i \in \Lambda^*$, no match combination is required at all, since the suffix arrays do not rely on tokenization, hence $t' := t_e \cdots t_f$ can be searched for as a single element in $O(|t'|)$ time.

Moreover, it is possible to define gaps of fixed length (measured in terms of number of T-characters, or alternatively, as number of A-annotations) between the individual elements, e.g. a pattern like $\sigma \overset{A:3}{\bowtie} \lambda$, indicating a distance of 3 arbitrarily A-annotated elements between σ and λ, can be evaluated in the same asymptotic time (because the length ℓ of the three wildcard elements following σ can be computed for each match candidate using `rank` and `select`, and then added to the candidate position, $q := p + |\sigma| + \ell$ used in (2) and (3) before the match range check for λ).

The property also holds when complex annotations and method 2 are used, at least when searching for prefixes of annotations, rather than arbitrary substrings of them. The distance calculations must then be made using the rank/select indexes for $B^{T\leftrightarrow A}$ to map positions between T and A, and those for $B^{A\leftrightarrow A}$ to compute the string length of annotations in A'. If arbitrary substring matching in annotations is required, the match process is delayed by a factor related to the length of λ, as every position inside the annotation must be checked for being a possible match continuation.

4 Complex Patterns

4.1 General patterns

Multiple annotation layers

It is straightforward to add further layers of annotation, e.g. semantic or morphological information, constituent classes etc. Each layer $A_1, A_2, \ldots$ is represented by an annotation string A'_i, a bitvector $B^{T\leftrightarrow A_i}$, and $B^{A_i\leftrightarrow A_i}$ if it is complex. Direct mappings between layers A_i, A_j are unnecessary, as they can be emulated using $B^{T\leftrightarrow A_i}$ and $B^{T\leftrightarrow A_j}$. Hence, total space consumption of the index grows in an additive manner as layers are added.

[5] The name *parallel suffix arrays* refers to the view of SA_T and $\mathrm{SA}_{A'}$ as parallel layers, both related to the same underlying text.

Branching patterns

An important step towards more powerful search patterns is the ability to process branching patterns, that is, patterns that specify multiple alternatives. This shall be denoted using a new operator $\oplus$, such that a pattern $\oplus(e_1, e_2, \ldots, e_m)$ is defined as matching all substrings of T that match any of the subexpressions e_i. If all e_i are distinct Σ-strings, the individual match sets for each e_i are disjoint, and the final result corresponds to the union set of the match ranges for the e_i.

But if some of the e_i refer to annotations or are themselves complex, i.e. sequential patterns or $\oplus$-expressions, the individual match sets might not be disjoint, causing the end result to contain duplicate matches, which makes it difficult to read and might cause frequency counts to be wrong. Hence, **duplicate elements must be detected** and removed from the individual match sets. This can be done either by creating a searchable result set representation, such as a hash table or tree, and inserting the matches one by one, rejecting matches that were inserted before; or, it can be done by creating a simpler, non-searchable result list and checking for each match for any e_i whether it is also a match for one of the other e_j, $j < i$. Both these methods are available when inverted files are used instead of suffix arrays, too, but if the second method is used, suffix arrays often have an advantage because the member check for the e_j, if it is a Σ- or Λ-string, involves only an $O(1)$ range check, whereas it would be logarithmic in an inverted file.

Sequences of complex elements

In section 3, the least-frequency property was established for sequential patterns, consisting of atomic elements and fixed-length-gaps, i.e. expressions like

$$e_1 \overset{Q_1:x_1}{\bowtie} e_2 \overset{Q_2:x_2}{\bowtie} \cdots \overset{Q_{m-1}:x_{m-1}}{\bowtie} e_m\,,$$

where $e_i \in \Sigma^*, \Lambda^*$; $Q_i \in \{\Sigma, \Lambda\}$; x_i integers. For even more powerful search patterns, it is important that the above can also be processed if the e_i are themselves complex, i.e. sequences or branching elements. This is indeed possible; the pattern then becomes a graph, and determining the least-frequent element, at which the matching should start, becomes a non-trivial problem. The number of matches of a sequence or branching subelement cannot be calculated accurately before the entire matching process has finished, but an upper bound can be determined: For a sequence, it is the frequency of its least-frequent subelement, for a branching element it is the sum of the frequencies of its branches. Based on this, it is possible to recursively determine the estimated best atomic subelement of the graph for the match combination process to begin. Once it has begun, the least-frequency property takes full effect during the processing of sequential substructures, and the range property accelerates the duplicate-checks where branching substructures are involved, as described above. Both is not true of inverted files, hence the theoretical performance of parallel suffix arrays is, generally, superior even for the most complex patterns.

Iteration

Another useful operator in powerful linguistic search patterns is the iteration operator, which is denoted by $\circledast(e)$ for any atomic or complex expression e. It corresponds to a sequence

$$e \overset{T:0}{\bowtie} e \overset{T:0}{\bowtie} \cdots \overset{T:0}{\bowtie} e$$

of undetermined length. Since all its elements are identical, the least-frequency property is preserved, even if the matching simply starts on the left end, or alternatively on the right end, and continues as long as new matches are found. Therefore, iteration elements can itself become part of complex patterns, and the three operations $\overset{Q:x}{\bowtie}$, $\oplus$ and $\circledast$ establish a pattern syntax with the power of regular expressions, over an annotated text with any number of annotation layers, and including fixed-length gaps (wildcards).

4.2 Gap-filling

In order to analyse linguistic patterns in specific contexts, it is desirable that not only substrings matching the entire pattern are identified, but that selected parts of the patterns, especially matches for gaps or annotation elements, can be extracted and separately returned as frequency lists. For example, if one wants to investigate the syntactic environment of "discussion", i.e. usages like "discussion on", "discussion with" etc., one might use a pattern like

$$\texttt{discussion} \overset{T:0}{\bowtie} \underbrace{\texttt{<Prep><Det>}}_{(*)} \overset{T:0}{\bowtie} \texttt{<N>}$$

and then obtain a frequency list of the content that matched the part marked by $(*)$. Parallel suf-

fix arrays are particularly well-suited for this purpose: Firstly, it is easy to keep track of the beginning and ending offsets of the desired subexpressions during the matching processing; secondly, frequency lists are easy to generate: Given starting positions p_1, p_2 of two matches for $(*)$, a comparison of invSA$[p_1]$ and invSA$[p_2]$ in $O(1)$ time suffices to determine their lexicographic order. Once the matches are in lexicographic order, identifying duplicates and counting the frequencies of distinct strings is easy.

4.3 Look-betweens and negation

Another feature related to gaps is the ability to define some of their content partially. The three types of patterns below are examples of this:

(a) $e_1 \overset{Q:x:y}{\bowtie} e_2$ (b) $e_1 \overset{Q:x:y}{\bowtie} [?e_3] e_2$ (c) $e_1 \overset{Q:x:y}{\bowtie} [!e_3] e_2$

(a) represents a gap of length $x \leq \ell \leq y$ elements on the annotation level Q; (b) requires that somewhere inside the gap there must be a match for e_3 (positive look-between); (c) means there must be no match for e_3 in the gap (negative look-between). Without going into further detail, it should be noted that these types of patterns can be incorporated into the matching process using match combination techniques similar to those described in section 3. There is, however, a specific disadvantage of suffix arrays when processing variable-length gaps $e_1 \overset{Q:x:y}{\bowtie} e_2$: Assuming that $\text{occ}(e_1) \leq \text{occ}(e_2)$, let (p, ℓ) be the position and length of a match for e_1. Let (q_i, p_i, ℓ_i) be a Q-annotation located at $p_i = p + \ell$, and

$$(q_{i+1}, p_{i+1}, \ell_{i+1}), \ldots, (q_{i+y}, p_{i+y}, \ell_{i+y})$$

the following y Q-annotations. Then we need to check whether a match for e_2 is found at any of the positions $p_{i+x}, \ldots, p_{i+y}$, which requires $\delta := y - x + 1$ look-ups in invSA$_Q$. Hence, the gap length variability δ becomes a factor in the time complexity of the match combination process. That is not the case when inverted files are used: It then suffices to check for matches at p_{i+x} and p_{i+y} stored in the inverted list for e_2. Since the inverted list is sorted, all other relevant matches must be located between these too and can be retrieved in one step.

5 Implementation and Results

5.1 Index construction and operation

The system has been implemented as a C++ program that takes as input a file containing the text T with three layers of annotations in XML: A_{POS} (POS-annotations); A_{lem} (baseforms of words, indexed using method 1 (see section 3); A_{cPOS} (POS along with morphological information; indexed using method 2).

The index construction is performed by first establishing the parallel layers and bitvectors and then creating SA$_Q$, invSA$_Q$ and, as an auxiliary data structure used to enable faster suffix array search, the wavelet tree WVT$_Q$ (Grossi et al., 2003) for each layer $Q \in \{T, A_{\text{POS}}, A_{\text{lem}}, A_{\text{cPOS}}\}$. For the construction of SA$_Q$, a multi-threaded version of the DC-algorithm (Kärkkäinen et al., 2006) is used, invSA$_Q$ is computed in a trivial way in one pass over SA$_Q$, and the wavelet tree WVT$_Q$ is constructed using a simple multi-threaded method (for details see Goller (2011)). The only highly time-consuming steps are the constructions of SA$_Q$ and WVT$_Q$. Their running times are given in Table 1.

For efficient pattern search, it is necessary to keep all data structures in main memory at all times. Compression methods for SA, invSA and WVT are available, cf. Navarro and Mäkinen (2007), but unfortunately, using them causes the time complexity of pattern search to be increased by a factor of $\Omega(\log n)$, eliminating the advantage it has over inverted files. As a result, using parallel suffix arrays requires a large amount of RAM. The implementation used to obtain the results described above was found to require $\approx (0.06 \cdot N)/1024$ MB for a corpus of N characters with the three annotation layers described above. Hence, on a 32-bit desktop computer with about 3 GB of memory available, a corpus of ≈ 52 million characters ($\approx$ 7 million words) can be processed efficiently. Therefore, although optimizing the implementation's use of RAM is certainly possible, it is quite clear that possibilities to use the described approach in linguistic practice depend on whether servers with sufficiently large RAM are available, and affordable.

5.2 Pattern Search

Table 2 presents response times for various kinds of patterns and illustrates, as expected, that the performance varies greatly depending on the complexity of the pattern; more specifically, it de-

	Threads Used	**Hard drive**	**Available RAM**	SA_T	WVT_T	SA_{POS}	WVT_{POS}	SA_{lem}	WVT_{lem}	SA_{cPOS}	WVT_{cPOS}
A	10	NFS	128 GB	3:17	3:48	1:30	1:13	1:27	11:46	3:03	2:03
B	20	Direct	512 GB	2:00	3:06	0:50	1:05	0:49	8:05	2:00	1:57
C	45	Direct	512 GB	2:00	2:37	0:51	0:54	0:55	6:16	1:49	1:43

Table 1: Construction times on three different system configurations. The text is 1.97 billion characters (375 million words) in length and contains approx. 27,000 distinct baseforms of words. Test A was performed on a server with AMD-Opteron CPUs 8356 (total 16 threads) and the hard drive mounted through NFS, tests B and C were conducted on a server with Intel Xeon X7560 processors (total 64 threads) and the hard drive installed locally. Time durations are given in the format h:mm.

	Pattern	**#results**	**Search time (ms)**	**Extraction time (ms)**
P1	millions	5,857	106	200
P2	thousands of	7,526	74	399
P3	#thousand# of	7,696	168	343
P4	discussion⟨IN⟩⟨NN⟩	1,296	213	80
P5	discussion$pr$$n$	1,894	372	118
P6	discussion[$pr$$n$]	1,894	530	111
P7	#preparation# $\overset{A_{POS}:0:2}{\bowtie}$ ⟨IN⟩ $\overset{T:0}{\bowtie}$ $\oplus$ (⟨NN⟩, ⟨NNS⟩)	752	191	28
P8	⟨JJ⟩⟨NN⟩⟨NN⟩ $\overset{A_{POS}:0:2}{\bowtie}$ ⟨IN⟩ $\overset{T:0}{\bowtie}$ $\oplus$ (⟨NN⟩, ⟨NNS⟩)	13,229	4,065	621
P9	⟨NN⟩⟨NN⟩ $\overset{A_{POS}:0:2}{\bowtie}$ ⟨IN⟩ $\overset{T:0}{\bowtie}$ $\oplus$ (⟨NN⟩, ⟨NNS⟩)	129,723	36,711	5,178

Table 2: Pattern processing times using hardware configuration A (see Table 1). Search time (identifying the set of match positions) and extraction time (extracting matches, but not including result printing). #..#-elements refer to A_{lem}, ⟨..⟩ to A_{POS}, $..$ to A_{cPOS}. Elements enclosed in [..] are marked for separate extraction and frequency counting (gap-filling). Times are in milliseconds.

pends on the "most specific atomic element" of the pattern. An element is atomic, if it refers to one layer (text or annotation) exclusively and contains no gaps. For example, the POS sequence ⟨JJ⟩⟨NN⟩⟨NN⟩ in P8, which would consist of three tokens in a standard inverted-file configuration, is atomic, as all three sub-elements refer to the same layer A_{POS} and can therefore be matched against $\text{SA}_{A_{\text{POS}}}$ in a single step. In accordance with the least-frequency property, the overall response time for the entire pattern depends on the number of occurrences of the most specific atomic element, which in this case is ⟨JJ⟩⟨NN⟩⟨NN⟩, rather than such high-frequency individual tokens as ⟨JJ⟩ or ⟨NN⟩. If the most specific atom is modified to be less specific, as in P9, the search time is increased by a factor of ≈ 9.

5.3 Discussion and Conclusion

The approach presented appears to be effective, especially for complex patterns that contain at least one relatively specific element. It provides efficient solutions for special tasks like context-specific pattern matching and frequency-list generation (described as gap-filling above), and it does not require any kind of tokenization, neither on the level of the main text, nor on the level of annotations and is hence suitable for corpora that involve annotations on the morpheme level, or across token boundaries, as well as for languages or writing systems that are hard to tokenize. Its biggest disadvantage is its high memory consumption, which however is likely to be less important in the future, as ever larger RAM hardware becomes available at increasingly low cost.

Although this has not been discussed in detail in previous sections, it is important to point out that the approach is *not* suitable in situations that call for frequent updates to the text or the annotations. The index structures described above, especially rank and select indexes for bit vectors as well as the suffix arrays themselves cannot be updated efficiently. Although data structures for suffix arrays that can be searched as well as dynamically updated are known, cf. (Russo et al., 2008; González and Navarro, 2008), using them would cause delays in the order of $O(\log n)$ (where n is the length of the text) in lookups of `select`, `rank` and SA, hence rendering the system considerably less efficient the corresponding version of an inverted file based system.

There are plans to release an open-source version of the implementation used for the tests described above as a corpus exploration tool for linguists before the end of the year.

References

Johannes Goller. 2011. *Exploring text corpora using index structures. PhD thesis*. To appear, Centrum für Informations- und Sprachverarbeitung, Ludwig-Maximilians-Universität München.

Rodrigo González and Gonzalo Navarro. 2008. Improved dynamic rank-select entropy-bound structures. In *LNCS 4957/2008, LATIN 2008: Theoretical Informatics*, pages 374–386, Berlin / Heidelberg. Springer.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. In *SODA '03: Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms*, pages 841–850, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Proc. of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554.

Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. 2006. Linear work suffix array construction. *J. ACM*, 53(6):918–936.

Gonzalo Navarro and Veli Mäkinen. 2007. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1):2.

Simon Puglisi, W. Smyth, and Andrew Turpin. 2006. Inverted files versus suffix arrays for locating patterns in primary memory. In Fabio Crestani, Paolo Ferragina, and Mark Sanderson, editors, *String Processing and Information Retrieval*, volume 4209 of *Lecture Notes in Computer Science*, pages 122–133. Springer Berlin / Heidelberg.

Luís M. Russo, Gonzalo Navarro, and Arlindo L. Oliveira. 2008. Dynamic fully-compressed suffix trees. In *CPM '08: Proceedings of the 19th annual symposium on Combinatorial Pattern Matching*, pages 191–203, Berlin, Heidelberg. Springer-Verlag.

P. Rychlý. 2007. Manatee/bonito – a modular corpus manager. In P. Sojka and A. Horák, editors, *First Workshop on Recent Advances in Slavonic Natural Language Processing 2007*, Faculty of Informatics, Masaryk University, Botanická 68a, 60200 Brno, Czech Republic.

Mikio Yamamoto and Kenneth W. Church. 1998. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27:28–37.