Parallel Suffix Arrays for Corpus Exploration*

Johannes Goller

Macmillan Limited
Chiyoda Building, 2-37 Ichigayatamachi,
Shinjuku-ku, Tokyo, 162-0843 Japan
jogojapan@gmail.com

Abstract. This paper describes how recently developed techniques for suffix array construction and compression can be expanded to bring a new data structure, called parallel suffix array, into existence, which is suitable as an in-memory representation of large annotated corpora, enabling complex queries and fast extractions of the context of matching substrings. It is also shown how parallel suffix arrays are superior to existing corpus search engines, in particular when sequential queries and corpora that are hard to tokenize are involved.

Keywords: Corpus search engine, suffix array, search engine for linguists

1 Introduction

Corpus search engines (CSE) are in high demand for a number of reasons: (a) They help corpus linguists to obtain statistics and usage information for words, and they do this with much higher accuracy and flexibility than web search engines (cf. Kilgarriff, 2007); (b) they enable the implementation of automated or interactive linguistic tests, for example to verify whether certain generalized syntactic constructions, e.g. represented by finite state grammars, do or do not occur in large corpus samples; (c) if sufficiently powerful, they transform an annotated text corpus into a tool for data mining (cf. Ananiadou, 2009).

CSEs accomplish these things in two steps: First, an index is constructed from annotated text, then queries are either manually inputted by users or generated by a separate process; the corpus search engine returns all matching substrings it finds for each query, either in random order, or sorted or grouped by user-defined criteria. It may also identify duplicate results and provide a frequency list of distinct matches. Existing CSEs differ from each other in several ways: (a) The kind of annotations they can process, (b) the complexity of queries, e.g. keyword search as opposed to regular expressions, (c) the data structures used inside the index. The choice of the data structure is significant, as it has a strong influence on how large the index grows in memory and how fast it can process queries. Fast query processing is important, esp. when large sets of auto-created query variations are processed, or when a multi-user system is designed to process many queries in parallel.

Section 2 gives an overview of existing CSEs. Their underlying indexes are all based on traditional relational databases or inverted file techniques. Section 3 describes *parallel suffix arrays*, a new data structure with very good theoretical performance for many types of complex queries, in many respects superior to existing data structures. Section 4 shows how the data structure can be expanded to a powerful CSE which will be called *sufex*. The final sections report on results of performance tests with a practical implementation of sufex and conclude with a brief discussion of its advantages and disadvantages.

* At the time when this article was prepared, the author worked at the Centrum für Informations- und Sprachverarbeitung, Ludwig-Maximilians-Universität München, Germany.

2 Existing corpus search engines

The simplest method of corpus search is to search the text linearly, for example using finite state tools like "grep". If the text contains annotations, e.g. in XML, and is very large, using such tools involves writing complicated regular expressions and is time-consuming. Somewhat more advanced are concordance-building environments such as "Wordsmith"[1] and user interfaces for finite state matchers, such as "Unitex"[2]. The query response time of these systems depends directly on the size of the text and may range from several minutes to several hours for corpora with millions of words.

An approach that uses a real index and provides much faster access is to fit the corpus and all annotations into a relational database, as described by Mark Davies (Davies, 2004; Davies, 2003) and available as an online search tool for various large corpora with POS-annotations[3]. The relational databases store k-grams such that there is a separate data column for every individual token (k columns in total), and the POS-tag of every token in a separate column (another k columns). Queries representing sequences of surface tokens and POS-tags, up to k in length, can be performed as SQL queries against the database. The search engine returns (a) the frequency of the k-grams that matched, and (b) the surface string itself, which can be used for a subsequent search on a full-text index of the surface string to retrieve match contexts. Although a detailed performance analysis is not available, practical tests of the online tool show response times in the subsecond to 2 seconds range on a 100-million-token corpus, including network delays and display time. The disadvantage of the approach, however, is obviously that it is limited to sequential queries of up to k tokens in length ("k-slots-queries"). It does not enable regular expressions (except within each slot) or gap-filling techniques (see 4.2), and it requires tokenized input.

The most advanced corpus search methods currently available make use of inverted file indexes, i.e. the data structures that web search engines are based on (Witten et al., 1999). The earliest documented example of this approach appears to be the "Corpus Query System" (Christ, 1994), which maps each token and each tag that occurs in the text to a list of the positions at which it was found, and performs lookups in these position lists to find query matches. This is essentially an inverted file architecture; it was upgraded several times and is currently known as the IMS Workbench[4]. Another CSE using inverted files is the "Linguist's Search Engine" (LSE) of the WebCorp project (Renouf et al., 2007). The system is still being tested, and there are no detailed reports on its performance. A third system, which has been available as open-source software for many years, is "Manatee" (Rychlý, 2000; Rychlý, 2007) and its (non-open-source) extension, the "SketchEngine" (Kilgarriff et al., 2004), of which an online demo exists[5].

Inverted-file based systems usually process simple queries in less than a second, even on very large corpora, and enable sequential queries and OR-queries of any length and including POS-tags or other elements that refer to pre-defined token-related annotations of the corpus. Like systems using the k-gram based approach described above, inverted-file based systems also require the corpus to be tokenized before they can index it.

The main advantages of the new approach described below are: (a) It does not require the text to be tokenized. Also annotations beginning or ending in the middle of tokens or spanning across multiple tokens are supported; (b) the system enables the full power of regular expressions and offers very good performance for *gap-filling* techniques (see in 4.2). The main disadvantage is its memory consumption; this and other pros and cons of the new approach are discussed in section 6.

[1] http://www.lexically.net/wordsmith/
[2] http://www-igm.univ-mlv.fr/~unitex/
[3] http://corpus.byu.edu/bnc/
[4] http://bulba.sdsu.edu/technical-manual.ps
[5] http://sketchengine.co.uk/

3 Parallel suffix arrays

3.1 Plain 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. In the context of corpus search systems, *the text* is the concatenation of all documents and all sentences of the entire corpus, which means billions of characters in the case of a corpus like the BNC[6].

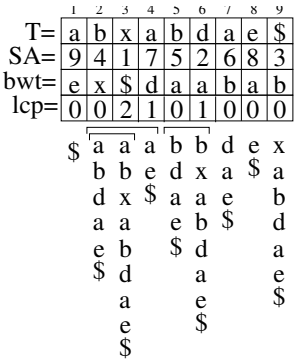

Figure 1: Suffix array SA for the string $T =$`abxabdae$`, along with auxiliary data structures *bwt*, *lcp* and "brackets" indicating the match ranges for substrings `ab`, `a` and `b`.

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.

For a long time, suffix arrays were not considered suitable for large-scale search problems, because no fast construction, retrieval or compression methods were available, so they were extremely hard to construct and use. During the past ten years, however, major progress was made in finding better construction and search algorithms: (a) **Linear Construction:** In 2003, three groups independently found methods to construct the suffix array in $O(n)$ time (Kim et al., 2003; Ko and Aluru, 2003; Kärkkäinen and Sanders, 2003); (b) **BW-Search:** a new and better search procedure (Ferragina and Manzini, 2000), based on a new auxiliary data structure, the *bwt* (Burrows Wheeler Transform); (c) **Wavelet trees:** a new data structure which enables efficient access to certain properties of *bwt* (Grossi et al., 2003) and increases the speed of BW-search; (d) **Various compression techniques** for SA, which enable more space-efficient storage; for an overview, see (Navarro and Mäkinen, 2007).

As a result of these improvements, it is possible to identify the match range for any pattern P, m characters in length, in $O(m)$ time[7] if BW-search and wavelet trees, but only weak compression are used. These new developments still have not solved the problem that searching the

[6] The British National Corpus, 100 million words, see `http://www.natcorp.ox.ac.uk/`

[7] Strictly speaking, the time is bound by $O(m(1+\log|\Sigma|/\log\log n))$, where $|\Sigma|$ is the size of the alphabet. However that is asymptotically equivalent to $O(m)$ 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), p. 42 for details.

SA (and *bwt*) requires frequent random access to large data structures, hence these must be stored in memory, while disk access would cause inacceptable delays. But since servers equipped with more than 100 GB of main memory have become available, it is reasonable to resonsider suffix arrays, and to increase their potential by adding features required for CSEs: annotation support and pattern matching. The following sections describe how that can be done.

3.2 Parallel suffix arrays

The first step is to allow annotations to enter the index. Suppose a text $T \in \Sigma^*$ of length n is given, and one layer of r annotations

$$A = ((a_1, p_1, \ell_1), (a_2, p_2, \ell_2), \ldots, (a_r, p_r, \ell_r))$$

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:

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

Although in this example the annotations each relate to one word of the text, they could as well start and end in the middle of words or span across several words. However, I do, at this point, assume that annotations are non-overlapping. There are two ways to bring these annotations into the 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 bwt and wavelet trees. 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 queries, i.e. queries containing both T-sequences and A-sequences.

Method 2: Complex annotations. Sometimes 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 ♯-symbols prevent undesired matches across annotation-boundaries.

How the bitvectors are used for mixed T/A' queries. Both the bitvector of method 1 and the bitvectors of method 2 need to undergo an indexation process, during which a *rank* index and a *select* index are generated for each bitvector, defined as follows: Let B be a bitvector and b its length, then $\texttt{rank}_B$ and $\texttt{select}_B$ are functions such that

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

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

$$\text{invSA}[j] := i \text{ s.t. } \text{SA}[i] = j.$$

The array invSA can be generated from SA in linear time.

To see how these data structures work together, consider a mixed query $\sigma\lambda$, 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 $\text{SA}_{A'}$. Clearly, the number of occurrences of σ in T is $occ_\sigma = r_\sigma - l_\sigma$, the number of matches for λ is $occ_\lambda = r_\lambda - l_\lambda$. 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 $p = \text{SA}_T[x]$ 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:

$$\begin{aligned}A\text{-element starts at p:} &\quad B^{T\leftrightarrow A}[p] = 1\\ \text{Position in } A'\text{:} &\quad p' := \texttt{rank}_{B^{T\leftrightarrow A}}(p + |\sigma|)\\ \text{Is } p' \text{ a } \lambda\text{-match:} &\quad l_\lambda \le \text{invSA}_A[p'] < r_\lambda\end{aligned}$$

If SA and invSA are available for random access, all of the above can be tested in $O(1)$ time, hence it takes $O(occ_\sigma)$ 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 $O(occ_\lambda)$). Hence it is possible to choose the matching direction according to whichever part of the query has fewer matches, i.e. let $occ_\mu := \min(occ_\sigma, occ_\lambda)$, then the match combination can be computed in $O(occ_\mu)$ time.

This is different when inverted files are used instead of suffix arrays, because they do not have the match range property. Verifying whether the position following a given match for σ is in the position list for λ requires a binary search of that list ($O(\log occ_\lambda)$ time), rather than a mere range check. Let $occ_M := \max(occ_\sigma, occ_\lambda)$, then the match combination generally takes $O(occ_\mu \log occ_M)$ time if inverted files are used.

This result can be extended to general sequential queries $\sigma_1\lambda_1 \cdots \sigma_m\lambda_m$ of any length m: The match combination time depends only on the least frequent (i.e. most specific) element if suffix

arrays are used, but it depends on the frequency of all elements if inverted files are used. In other words, suffix arrays have a **least-frequency property** for match combinations.

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 query 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 $p + |\sigma| + \ell$ before making 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.

I decided to name these data structures and methods **parallel suffix arrays**, because they are based on the idea to consider T and A' as parallel (and to connect them through indexed bitvectors). Further generalization of this idea leads to a complete CSE which I will call **sufex**[8].

4 Sufex

4.1 General patterns

Multiple annotation layers It is straight-forward 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.

Branching queries When defining queries to a CSE it is important to be able to state alternatives to parts of the query. For example, one element of a larger query may be an expression that is supposed to either match a word annotated as verb, or *alternatively*, a form of "to be", followed by a participle (to indicate a verb in passive voice). Generally speaking, a query operator for alternatives, such as $\oplus(e_1, e_2, \ldots, e_m)$, is required, defined to return the set of substrings of T that match one of the subexpressions e_i. In traditional inverted-file based systems this is usually written as e_1 `OR` $e_2 \cdots$ `OR` e_m. If all e_i are distinct Σ-strings, the individual match sets for each e_i are disjoint, and computing the final result set is simply a matter of concatenating them.

But if some of the e_i refer to annotations or are themselves non-atomic, i.e. sequences 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 each of the matches unless it has been 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 have an advantage because the member check for the e_j, if it is atomic, involves only an $O(1)$ range check, whereas it would be logarithmic for an inverted file.

[8] For **suf**fix-array based ind**ex**.

Sequences of complex elements In section 3, the least-frequency property was established for sequential queries, 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. To reach the full power of regular expressions, it is important that such queries can also be processed if the e_i are themselves complex, i.e. sequences or branching elements. This is indeed possible; the query 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 approximately 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 the case with inverted files, hence the theoretical performance of sufex is, generally, superior even for the most complex query structures.

Iteration The final operation required for regular expressions 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 queries, and the three operations $\overset{Q:x}{\bowtie}$, $\oplus$ and $\circledast$ establish a query language with the power of regular expressions, over an annotated text with any number of annotation layers.

4.2 Gap-filling

In order to analyse linguistic patterns in specific contexts, it is desirable that not only substrings matching the entire query are identified, but that selected parts of the query, 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 issue a query such as

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

and then obtain a frequency list of the content that matched the query part marked by $(*)$. The sufex matching process is particularly well-suited for this purpose: Firstly, it is easy to keep track of the beginnings and ends of the desired subexpressions during the matching; secondly, frequency lists are easy to generate: Given starting positions p_1, p_2 of two matches for the subexpression, a comparison of invSA$[p_1]$ and invSA$[p_2]$ in $O(1)$ time suffices to determine their lexicographic order. Moreover, if the *lcp*-array has been generated, duplicate checks can be performed in $O(1)$ time, too (as described by Goller, 2010).

4.3 Look-betweens and negation

Another feature related to gaps is the ability to define some of their content partially. For example, Sufex enables queries of the three types below:

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

(a) represents a gap of length $x \leq \ell \leq y$ elements of the annotation level Q; (b) specifies 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 queries can be readily incorporated into the matching process.

Table 1: Construction times for a sufex with three annotation layers and texts of various sizes. A thread number of "10/14" means that 10 threads were used during SA construction, but 14 for the construction of WVT. Time durations are in the format h:mm.

Corpus	**#chars (millions)**	**Size Λ_{lem}**	**Threads**	SA_T	WVT_T	SA_{POS}	WVT_{POS}	SA_{lem}	WVT_{lem}	SA_{cPOS}	WVT_{cPOS}
S8	589	61,394	10/14	1:00	1:08	0:26	0:23	0:25	3:56	0:53	0:40
Reuters	958	93,018	10/14	2:06	1:42	0:45	0:34	0:44	5:48	1:31	1:02
Prod22	1,206	86,433	10/14	2:15	2:14	0:57	0:42	0:47	7:12	3:00	2:10
S5-8	1,966	72,036	10	3:17	3:48	1:30	1:13	1:27	11:46	3:03	2:03

5 Sufex in practice

5.1 Construction

Sufex was implemented as a C++ program which takes as input a configuration file stating the number and types of annotation layers, and a file containing the text T with annotations in XML. In all tests reported here, three annotation layers were indexed: A_{POS} (POS-annotations); A_{lem} (baseforms of words, mapped to a 32-bit single-integer alphabet Λ_{lem}); A_{cPOS} (POS along with morphological information; implemented as complex annotations).

The index construction is performed by first establishing the parallel layers and bitvectors and then creating SA_Q, invSA_Q and the wavelet tree WVT_Q 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 for WVT_Q the multi-threaded construction method described by (Goller, 2010) is used. All these steps take less than 15 min each even for corpora with billions of characters, except the constructions of SA_Q and WVT_Q. The running times for the latter two are given in table 1.

All tests were carried out on a large server with 4 AMD-Opteron CPUs (16 cores in total) and 128 GB of main memory. When the entire index is in RAM, total memory consumption was found to be 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 server with about 3 GB of memory available for use by sufex, a corpus of ≈ 52 million characters (≈ 7 million words) can be processed efficiently[9].

5.2 Query processing

Table 2 shows the processing times of several example queries. It demonstrates very clearly the effects of the least-frequency property: Queries that contain at least one relatively rare element, such as a specific word or lemma, are processed in much less than a second, whereas a query with atomic elements that are all very frequent takes much longer.

[9] Note that during index construction, memory consumption is lower, because while one annotation layer is created, the others need not be present in memory.

Table 2: Query processing times for the S5-8 corpus (approx. 2 billion characters), including search (identifying the set of matching positions) and extraction (extracting matching strings), but not the actual printing of the results on screen. #..#-elements refer to A_{lem}, $\langle..\rangle$ to A_{POS}, \$..\$ to A_{cPOS}. Subqueries enclosed in [..] are marked for separate extraction and frequency counting (gap-filling). Processing times are given in milliseconds.

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

When comparing these numbers to those of web search engines, it is important to remember that web search engines are designed to return only the N best results, typically $N \approx 2000$. For the purpose of linguistic investigations, the user is generally interested in retrieving the complete result set, even if it means waiting for several minutes. But if necessary, partial result set retrieval could be implemented in sufex, and it would result in a significant reduction of response times.

6 Conclusion

Clearly sufex is able to process large annotated corpora and enable excellent query response times, which are generally in the subsecond range even for complex queries with many annotation elements, as long as the query contains at least one specific subelement. In this respect, sufex is superior to inverted file-based CSEs. Another important advantage is that neither annotation boundaries nor query elements need to be related to any kind of pretokenization of T. Note that this is useful not only when searching inside words, but also when searching for sequences of words or annotations: As long as they are all on the same layer, they can be searched for directly in SA, and no match combination process is required.

Drawbacks of the approach include: (a) Annotations must be defined as layers. Overlapping annotations are only possible if located on different layers. (b) High performance is only possible if the main data structures are loaded into RAM. This means that, at the present time, sufex-based CSEs for corpora with hundreds of millions of tokens require a relatively high, but not unrealistic, hardware investment. However, given its performance and flexibility, sufex is well-suited for projects involving larger user-groups and joint research efforts of multiple institutions, which makes the availability of the necessary hardware budget more likely.

Sufex' gap-filling capabilities (4.2) allow it to be used not only interactively by linguists, but also by automated bootstrapping processes. For example, given a set S of words or patterns of interest, submitting a query $g_L s g_R$ for each $s \in S$, where g_L and g_R are "gap-like" wildcard expressions for the left and right context, and having sufex generate frequency lists of the content found for g_L and g_R, makes it possible to iteratively refine a set of contexts, which in turn can be used to find further elements belonging to S. While such bootstrapping has already been used in many areas of NLP research, sufex improves it in two ways: (a) Contexts can be defined as complex expressions, including annotation symbols, rather than mere *windows of N words*; (b) Sufex is an index and provides answers within milliseconds (the more specific the queries become in the process, the faster the retrieval, due to the least-frequency property). It therefore has the potential to greatly facilitate the further development of such bootstrapping methods.

References

Ananiadou, S. (2009). Text mining for biomedicine. In Prince, V. and Roche, M., editors, *Information Retrieval in Biomedicine: Natural Language Processing for Knowledge Integration*, pages 1–10. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA.

Christ, O. (1994). A modular and flexible architecture for an integrated corpus query system. In *Proceedings of COMPLEX'94, 3rd Conference on Computational Lexicography and Text Research, July 7-10*, pages 23–32, Budapest, Hungary.

Davies, M. (2003). Relational n-gram databases as a basis for unlimited annotation on large corpora. In *Proc. of Workshop on Shallow Processing of Large Corpora, March 27*, pages 23–33, Lancaster, UK.

Davies, M. (2004). A match made in corpus heaven: the bnc and wordnet in relational database form. In *25th Conference of the International Computer Archive of Modern and Medieval English*.

Ferragina, P. and Manzini, G. (2000). Opportunistic data structures with applications. Technical report, University of Pisa, TR-00-03.

Goller, J. (2010). *Exploring text corpora using index structures. PhD thesis*. To appear, Centrum für Informations- und Sprachverarbeitung, Ludwig-Maximilians-Universität München.

Grossi, R., Gupta, A., and Vitter, J. S. (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.

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

Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. In *Proc. 13th International Conference on Automata, Languages and Programming*. Springer.

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

Kilgarriff, A. (2007). Googleology is bad science. *Comput. Linguist.*, 33(1):147–151.

Kilgarriff, A., Rychlý, P., Smrz, P., and Tugwell, D. (2004). The sketch engine. In Williams, G. and Vessier, S., editors, *Proceedings of the Eleventh EURALEX International Congress*, France. Faculté des Lettres et des Sciences Humaines, Université de Bretagne Sud.

Kim, D. K., Sim, J. S., Park, H., and Park, K. (2003). Linear-time construction of suffix arrays. In *Proc. 14th Ann. Symposium on Combinatorial Pattern Matching*, Berlin / Heidelberg. Springer.

Ko, P. and Aluru, S. (2003). Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, Berlin / Heidelberg. Springer.

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

Renouf, A., Kehoe, A., and Banerjee, J. (2007). Webcorp: an integrated system for web text search. In Nesselhauf, C., Hundt, M., and Biewer, C., editors, *Corpus Linguistics and the Web*. Rodopi, Amsterdam.

Rychlý, P. (2000). *Korpusové manažery a jejich efektivní implementace (engl.: Corpus Managers and their Effective Implementation)*. PhD thesis. Faculty of Informatics, Masaryk University, Brno, Czech Republic, `www.fi.muni.cz/~pary/disert.ps`.

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

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA.