DESIGNING EFFICIENT ALGORITHMS FOR QUERYING LARGE CORPORA

PAUL MEURER

University of Bergen

ABSTRACT

I describe several new efficient algorithms for querying large annotated corpora. The search algorithms as they are implemented in several popular corpus search engines are less than optimal in two respects: regular expression string matching in the lexicon is done in linear time, and regular expressions over corpus positions are evaluated starting in those corpus positions that match the constraints of the initial edges of the corresponding network. To address these shortcomings, I have developed an algorithm for regular expression matching on suffix arrays that allows fast lexicon lookup, and a technique for running finite state automata from edges with lowest corpus counts. The implementation of the lexicon as suffix array also lends itself to an elegant and efficient treatment of multi-valued and set-valued attributes. The described techniques have been implemented in a fully functional corpus management system and are also used in a treebank query system.

[1] INTRODUCTION

In this article, I describe several new efficient algorithms for querying large annotated corpora, among them a technique for running finite state automata from edges with lowest corpus counts (described in Section [4]), and an implementation of regular expression matching on suffix trees for fast lexicon lookup (described in Section [3]).

The algorithms were designed and implemented (in the Common Lisp programming language) during the development of *Corpuscle*¹ (Meurer 2012a), a corpus query engine and corpus management system that is in use for a variety of corpora of different types. Some of the algorithms are also used in *INESS-Search* (Meurer 2012b), a treebank query system that is part of the *INESS*² treebanking infrastructure.

An evaluation of our system against *Corpus Workbench* using the Norwegian Newspaper Corpus and other large corpora shows that our system is as fast or

[1] See <http://clarino.uib.no/corpuscle>

[2] See <http://clarino.uib.no/iness>

significantly faster on most types of queries. Comparative benchmark results for typical queries have been published in (Meurer 2012a).

[2] CORPUS DESIGN PRINCIPLES

In order to set the stage for the algorithms to be described, we shortly outline the design principles that underlie the Corpuscle query engine. In many respects, the design is quite similar to that of Corpus Workbench, but there are also important differences.

In a technical sense, a corpus can be described as a sequence of corpus positions that are annotated with values for one or several (positional) attributes. The words of the underlying texts comprise the mandatory attribute *word*, that is, a minimally annotated corpus consists of a sequence of words. Other possible attributes may encode grammatical features (part of speech, morphosyntax etc.), other linguistic annotations and metadata. Attribute values can be atomic or structured. Corpuscle has built-in support for structured attributes, both for multi-valued and set-valued attributes and combinations thereof.

In addition to corpus positions that encode word tokens together with their annotations, Corpuscle also uses corpus positions to encode structural data.³ Here, Corpuscle differs from CWB, where structural attributes are tied to the corpus position of the following word token. The advantage of giving structural data their own dedicated corpus positions is twofold: First, the original order of consecutive XML tags can always be unambiguously reconstructed from the encoded corpus. And second, structural data can also have positional attributes. This can be an advantage when attributes that scope over more than a single corpus position are coded as positional attributes.⁴

Corpuscle uses, as CWB and Manatee do, a static, file based representation of the attribute lexica (the set of strings that do occur as attribute values for a given attribute). A lexicon file implements a mapping between attribute values and numerical value IDs, and vice versa. In Corpuscle, I have chosen Suffix arrays as the data structure to implement the lexica. We will see in Section [3] that suffix arrays are well suited for the implementation of regular expression matching in the lexicon. In addition, an inverted index encodes the mapping from attribute values to the sets of numerical corpus positions where those values are taken. Those two files in tandem allow one to search for corpus positions that match a given attribute value, or, more generally, a boolean combination of regular expressions in corpus attributes.

When a query comprises more than one corpus position, the query evaluation

[3] Since structural data is represented in the vertical file as XML start, end and empty tags, possibly containing arbitrary XML-style attribute-value-pairs, I will use the terms XML tag and structural attribute (value) synonymously.

[4] As a consequence, the query language of Corpuscle has a semantics that differs slightly from that of CQP.

algorithm first tries to find all corpus positions that match the query constraints for one of the positions, and then checks the contexts of all matching corpus positions. This look-up is accomplished via an attribute corpus file for each attribute containing a vector of the value IDs of all corpus positions.

An additional file, the structure tree file, encodes the structure of the structural attribute. The structure tree contains for every corpus position a pair $(p_l(c), p_r(c))$ of pointers that point to the structural element that immediately contains the position. More exactly, if c is a start tag position, $p_l(c)$ points to the start tag of the enclosing element and $p_r(c)$ points to the corresponding end tag; if c is an end tag position, $p_l(c)$ points to the corresponding start tag and $p_r(c)$ points to the end tag of the enclosing element; otherwise, $p_l(c)$ and $p_r(c)$ point to the start and end tags of the enclosing element.

Using the structure tree, it is straightforward to navigate in the XML tree of the underlying texts and to calculate sentence and other contexts for a given corpus position. The structure tree is also consulted when it has to be checked whether matches of balanced tags in a query correspond to balanced tags in the corpus.

The corpus index files are calculated from a vertical file that has a format similar to CWB's vertical file. Each line in the vertical file corresponds to a corpus position, and the values of the positional attributes that are defined for the given corpus are separated by *tab* characters. Structural positions are represented by XML-style tags that may include attribute-value pairs. They, too, can have positional attribute values.

[3] REGULAR EXPRESSION MATCHING IN THE LEXICON: SUFFIX ARRAYS *The lexicon*

In both the on-disk corpus representation and in the inverted index, which associates to every attribute value the list of corpus positions where that value is taken, values are represented as numeric (natural number) identifiers (IDs). In order to be able to convert back and forth between ID and corresponding value, a table has to be maintained that maps values to IDs and vice versa. This table, which is called the attribute lexicon, can be organized in different ways. In the most basic implementation, the mapping table is kept in memory, where it can easily be implemented as a hash table (mapping values to IDs) combined with a vector of values (mapping IDs to values).

However, except for small corpora, it is unfeasible to store the entire lexicon in main memory. Even if enough memory is available, loading the lexicon in the startup phase of the corpus software can take a long time because the whole lexicon table structure has to be built, and (in a Lisp system) this large structure is visible to the garbage collector, making full GCs time consuming. A better approach is to organize the lexicon in a file and map that file into main memory us-

ing the Unix system call *mmap* (or a similar device for memory-mapped file I/O). Since *mmap* implements demand paging, only those parts (pages) of the lexicon file that are actually needed are loaded into main memory in a lazy manner.

A convenient way of storing the lexicon of a given corpus attribute on disk is to concatenate the sorted or unsorted list of value strings to a long string, using a character not otherwise occurring in the values (e.g. *Null* or *Newline*) as separator. Here, the question arises how an efficient mapping between string values and IDs should be implemented. Since the lexicon values have different length, they cannot any more be accessed via an offset that is proportional to the value's ID, as was the case in the basic in-memory implementation. Instead, an auxiliary file has to be maintained that maps IDs to offsets in the lexicon file.

The inverse mapping from values to IDs can be realized in different ways. An efficient way is to store the mapping table as a disk-based btree or a similar data structure, depending on the availability of implementations for the chosen programming language. Even a relational database could serve as lexicon index.

In addition to the lexicon files, a file that maps corpus positions to value IDs, the corpus file, has to be maintained for every corpus attribute. This file, which is structured as a simple array, is necessary to reconstruct the attribute values for a given corpus position, both when doing a linear scan of the corpus, which is necessary in certain types of queries, and to reconstruct a segment of the original annotated text for display purposes.

Regular expressions

One important function of the lexicon is to serve as an index for corpus queries, notably for queries involving regular expressions. If one relies on existing regular expression packages that operate on strings (or memory-mapped files) without preprocessing them, regular expression matching is very easy to implement. A regular expression query (a query that tries to find all attribute values that match a given regular expression) can be run on the entire lexicon to find all matching values. It has to be made sure that valid matches coincide with attribute values; they must not stretch over more than one value (i.e. contain the separator character), and they should start and end in the separator character. In order to be able to obtain the found value's ID immediately from the lexicon, the ID can be integrated into the lexicon; it may be prepended to the value in a fixed byte-size representation for easy and fast access.

This straightforward approach to regular expression matching is taken in most corpus engines, e.g. in Corpus Workbench and Manatee. The disadvantage of such an approach is that evaluating a regular expression is linear in time; the whole lexicon has to be processed sequentially to find all matches. Even though good regular expression algorithms that operate on strings use optimizations like Boyer-Moore string matching and similar techniques that make them fast for many types

of regular expressions (those that contain a long enough literal substring), they are not able to exploit the potential that lies in preprocessing the lexicon string.

As a first step to accelerate certain regular expression queries one could try to use the btree (or similar) index that is used to match values to IDs. Such an index allows finding all strings that have a given prefix in constant time. This approach, however, does not help to accelerate regular expressions that search for strings that contain a substring at an arbitrary position/offset. Data structures that allow one to find arbitrary substrings in constant time do exist; among them are Patricia trees, suffix trees and suffix arrays. Suffix arrays are the most space efficient. They are also easiest to construct if one does not need to build them in linear time. For this reason, I have chosen suffix arrays as the data structure to encode the lexicon. As we will see, suffix arrays lend themselves to an efficient implementation of regular expression matching.

Suffix arrays

Suffix arrays are a remarkably simple, yet very powerful data structure. They were developed by Manber & Myers (1993) as a space-efficient alternative to Suffix trees.

A Suffix array indexing a string L (the lexicon in our case) is an array S of string positions $p_i, i = 0, \dots, l - 1$ (where l is the string size) that represents the suffixes⁵ of the string in lexicographic order. Given array indices i and j with $i < j$, the suffix starting at p_i lexicographically precedes the suffix starting at p_j . A Suffix array allows one to locate the position(s) of an arbitrary substring of L very efficiently. Since the suffixes are ordered lexicographically, the positions of a given substring s , which amounts to the positions of all suffixes having s as prefix, can be found using binary search. With a naïve algorithm, search times of $\mathcal{O}(m \log n)$ can be achieved, where n is the size of the lexicon, and m is the length of the string to search for. More sophisticated algorithms that use auxiliary data structures (LCPs, longest common prefixes) achieve $\mathcal{O}(m + \log n)$.

From this it is apparent that suffix arrays are well suited as indices for simple dictionary lookup.

Regular expressions with suffix arrays

The key idea behind using suffix arrays as an efficient index for regular expression evaluation is to identify literal substrings in a given regular expression that can be looked up in the suffix array of the string to be searched. In the case of lexicon lookup, those literal substrings can be used as a filter to identify match candidates which have to be checked by the full regular expression.

Baeza-Yates & Gonnet (1996) have used similar ideas to implement regular expression search on preprocessed text using Patricia trees.

[5] The suffix of a string L at position i is the substring of L from position i to the end of the string.

The regular expressions considered here are supposed to match substrings of lexicon values; the expression ab for instance matches all lexicon values that contain the substring ‘ ab ’. To match whole prefixes or suffixes of lexicon values, the left (e.g. ‘ $^$ ’) or right (e.g. ‘ $^$ ’) value separator character has to be included in the regular expression. The expression ab for instance matches all values starting with ‘ ab ’.

Pre-filtering

To simplify the argument, I consider only regular expressions R that are expressed as a combination of concatenation (written as juxtaposition of subexpressions), union ($|$), Kleene star ($*$) and Kleene plus ($^+$), with letters $a \in \Sigma$ and the any-symbol (Σ , = union of all letters) as basic building blocks. I will describe an algorithm that converts each such regular expression R into a parameterized set of filter expressions $F^l = F^l(R)$ with the following properties:

- (1) • each $F = F^l$ is a disjunction of sets of strings:

$$F = \bigcup F_j, F_j = \{s_{j0}, \dots, s_{jn_j}\};$$
 - each match of R contains all the strings s_{ji} of one disjunct F_j as substrings.

The parameter l roughly corresponds to the maximal length of the substrings s_{ji} for each j .

Once a filter $F(R)$ is computed, the regular expression R can be evaluated as follows:

- For each disjunct F_j , find all lexicon values v that contain each of the strings $s_{ji} \in F_j$ as substrings. This is most efficiently done by setting a bit in a bit vector B_{ji} of lexicon size for each value that contains s_{ji} . When value IDs are coded into the lexicon as described above, those IDs are immediately available by suffix array lookup. From those B_{ji} , a bit vector B_j is calculated by bitwise *logand* that codes the values matching F_j .
- Bitwise *logor* of all B_j results in a bit vector B that codes the IDs of all values matching the filter F .
- In certain cases, the mapping $R \mapsto F(R)$ is *strict*, which means that the values matching $F(R)$ are *exactly* the values matching R . We will see below that regular expressions composed solely of characters, concatenation and union naturally give rise to strict filters.
- In most other cases, the values matching $F(R)$ will have to be matched against the regular expression R in a last step.

Given a regular expression R , there are two factors that determine the efficiency of the algorithm outlined above: The number of strings s_{ji} in F^l for a given value of the parameter l (abbreviated as $|F^l|$), and the specificity of F^l ($spec(F^l)$), i.e. the ratio of the number of values matching R to the number of values matching F^l (which is a number between 0 and 1). For an efficient filter F , $|F|$ should be low and $spec(F)$ should be high. These numbers cannot however be controlled independently; a highly specific filter generally entails many disjuncts F_j , and a filter with few disjuncts gives rise to low specificity.

It is difficult to strictly calculate the optimal value of l ; the value of $l = 3$ used in the code was determined by experiment.

The filter construction algorithm

Consider the following expansion operations on regular expressions:

- (2) Associative expansion:

$$\alpha_r : A(X|Y) \mapsto AX|AY$$

$$\alpha_l : (X|Y)B \mapsto XB|YB$$

- (3) Kleene plus expansion:

$$\pi_r : aX^+ \mapsto aXX^*$$

$$\pi_l : X^+b \mapsto X^*Xb$$

- (4) Kleene star expansion:

$$\sigma_r : aX^* \mapsto aXX^*|a$$

$$\sigma_l : X^*b \mapsto X^*Xb|b$$

where A, B, X, Y are regular expressions and a, b letters of the alphabet. These expansions operate on subexpressions of regular expressions, e.g.:

- (5) $ab(c|d)ef \mapsto \alpha_r[ab(c|d)]ef = (abc|abd)ef$

Application of any of the expansions does not change the set of values matched by the regular expression.

I call the toplevel set of all Kleene star subexpressions the *cyclic part* of a regular expression. It is easy to see that by applying the expansions (2) and (3) recursively to a regular expression R we can arrive at an expression that is a disjunction of regular expressions:

- (6) $\bar{R} = R_0|R_1|\dots|R_m$

such that each R_j is disjunction-free and contains no Kleene pluses except in the cyclic part.

As an example, consider the regular expression (7):

$$(7) \quad a(b(c|d^+)^+\Sigma^*e|fg)h$$

Application of α_l and α_r gives

$$(8) \quad ab(c|d^+)^+\Sigma^*eh|afgh$$

and application of π_l results in

$$(9) \quad ab(c|d^+)(c|d^+)^*\Sigma^*eh|afgh.$$

In the last step, a new disjunction and a new Kleene plus were introduced in the acyclic part, so expansions (2) and (3) can be applied again to give (10) and (11).

$$(10) \quad abc(c|d^+)^*\Sigma^*eh|abd^+(c|d^+)^*\Sigma^*eh|afgh$$

$$(11) \quad abc(c|d^+)^*\Sigma^*eh|abdd^*(c|d^+)^*\Sigma^*eh|afgh$$

The expression (11) is now of the form (6) with $R_0 = abc(c|d^+)^*\Sigma^*eh$, $R_1 = abdd^*(c|d^+)^*\Sigma^*eh$, $R_2 = afgh$.

Each R_j can be written in the following form:

$$(12) \quad R_j = C_{j0}s_{j0}C_{j1}s_{j1}\dots C_{jn_j}s_{jn_j}C_{jn_j+1}$$

where the C_{ji} are Kleene star expressions (non-empty except for possibly C_{j0} and C_{jn_j+1}), and the s_{ji} are (possibly empty) character strings. In our example (11), we have⁶

$$(13) \quad R_1 = C_{10}s_{10}C_{11}s_{11}C_{12}, \\ C_{10} = \varepsilon, s_{10} = abd, C_{11} = d^*(c|d^+)^*\Sigma^*, s_{11} = eh, C_{12} = \varepsilon.$$

Observe that a match of R is a match of $\bar{R}$ and a match of at least one of the disjuncts R_j . Furthermore, a match of R_j has to contain all of the strings s_{ji} as substrings. Thus, by setting $F_j = \{s_{j0}, \dots, s_{jn_j}\}$ (disregarding empty strings) and $F = \bigcup F_j$ we have constructed a filter F with the desired properties (1). For our example, we get:

$$(14) \quad F_0 = \{abc, eh\}, F_1 = \{abd, eh\}, F_2 = \{afgh\}.$$

In the application of the Kleene plus expansion rule (3), there is in most cases a choice between left (π_l) and right (π_r) expansion. Since the goal is to construct a filter that is as specific as possible, we always choose the expansion in the direction of the longest adjacent substring, that is, in a situation like

$$(15) \quad \dots C_0s_0X^+s_1C_1\dots$$

[6] ε denotes the empty string.

we choose π_l if $|s_0| > |s_1|$ and π_r otherwise. Here, C_0 and C_1 stand for sequences of Kleene plus or Kleene star expressions. When $|s_0| > |s_1|$, the expansion will be

$$(16) \quad \dots C_0 s_0 X X^* s_1 C_1 \dots$$

If X itself is a string s or has a string prefix ($X = sY$), the expansion results in a longer string $\bar{s}_0 = s_0 s$ as part of the filter, and thus a more specific filter. This is not guaranteed, but the criterion is at least a reasonable heuristic.

In cases where an expanded disjunct R_j contains Kleene stars, one can try to expand these using the Kleene star expansion rules (4) in order to increase the length of the longest string s in the disjunct. This, however, comes at a price, because (4) introduces a new disjunct on each application. So we extend the algorithm by applying either σ_l or σ_r to the Kleene star subexpression X^* adjacent to the longest substring s_{ji} in (13) (if there is one), depending on whether the longest substring is to the left or to the right of X^* . Again, it is not guaranteed that this process will lead to a longer maximal substring (e.g. $aa(\Sigma^*b)^*c$).

After the application of (4), rules (2) and (3) have to be applied again to expand newly created toplevel disjunctions and Kleene pluses.

I define the series of filters F^l , $l = 1, 2, 3, \dots$ by letting each F^l be the result of the iterative application of the expansions (2) – (4) until the longest string in each F_j^l has length greater or equal l , or until no rule is applicable. It can be shown that this iteration always comes to a halt.

Consider as an example the regular expression (17).

$$(17) \quad ab^*c$$

The successive expansions and filters of (17) are given in (18).

$$(18) \quad \begin{aligned} \bar{R}^1 &= ab^*c, F^1 = \{a, c\} \\ \bar{R}^2 &= ab^*bc|ac, F^2 = \{a, bc\}, \{ac\} \\ \bar{R}^3 &= ab^*bbc|abc|ac, F^3 = \{a, bbc\}, \{abc\}, \{ac\} \\ \bar{R}^4 &= ab^*bbbc|abbc|abc|ac, F^4 = \{a, bbbc\}, \{abbc\}, \{abc\}, \{ac\} \end{aligned}$$

This example would suggest that a more balanced expansion could be desirable. The first filter disjunct in all F^l contains the one-character string a , which means that all values containing a have to be looked up in the lexicon and the bits for all those values have to be set in the corresponding bit vector in the construction of F^l . If one instead expanded $\bar{R}^3$ further using the left-side rule σ_l , one would get

$$(19) \quad \bar{R}^{4'} = abb^*bbc|abbc|abc|ac, F^{4'} = \{ab, bbc\}, \{abbc\}, \{abc\}, \{ac\}.$$

Since in general many fewer values contain ab than a , $F^{4'}$ should be more efficient than F^4 . There is however not enough to be gained in practice to make it worth while implementing the more complicated algorithm.

Improvements: string order, strict filters, anchored expressions

It might strike the reader that the order of the strings s_{ji} in the filter disjuncts F_j is not used in the application of the filters. Taking the ordering into account in the filtering would however make it impossible to use the sketched efficient bitvector technique to calculate match candidates. Instead, a table of matching values together with match offsets would have to be maintained in every stage of the evaluation of a filter disjunct, in order to exclude those candidate matches for which the strings s_{ji} match in the wrong order or overlap. Experiments have shown that the cost of maintaining this table is much higher than the efficiency gain from not having to run the full regular expression on those candidates.

It is clear that regular expressions R that are composed of characters from the alphabet, concatenation and union alone give rise to strict filters. In deriving $\bar{R}$, only the associativity expansion rule is used, and $\bar{R}$ and F are of the form (20) and (21).

$$(20) \quad \bar{R} = s_0 | s_1 | \dots | s_m,$$

$$(21) \quad F = \{s_0\}, \{s_1\}, \dots, \{s_m\}.$$

There is a special case where *anchored* regular expressions (expressions containing the left or right boundary mark) lead to strict filters: those that after associativity expansion consist of disjuncts of the form

$$(22) \quad {}^\wedge s_l \Sigma^* s, \quad s \Sigma^* s_r \$, \quad {}^\wedge s_l \Sigma^* s \Sigma^* s_r \$ \quad \text{or} \quad {}^\wedge s_l \Sigma^* s_r \$.$$

The strings s_l , s and s_r have to satisfy in addition the conditions that s is not a substring of s_l or s_r , that no suffix of s_l is a prefix of s , no suffix of s is a prefix of s_r (where applicable), and no suffix of s_l is a prefix of s_r (in the last expression). These conditions are necessary to make sure that string matches in the filter do not overlap. The correct ordering is automatically satisfied because the boundary characters cannot occur inside values.

Application: querying set-valued and multi-valued attributes

There is one obvious additional type of regular expressions with strict filters: those that exactly match the matching values of a given filter $F = \bigcup F_j$, $F_j = \{s_{j0}, \dots, s_{jn_j}\}$. If we in addition demand that no two strings s_{ji} and s_{jk} from the same disjunct F_j overlap, the regular expression with strict filter F_j can be written as (23).

$$(23) \quad \bar{R}_j = \bigcup_{\{s_0, \dots, s_{n_j}\} \in \Pi(F_j)} s_0 \Sigma^* s_1 \Sigma^* \dots \Sigma^* s_{n_j}$$

That is, $\bar{R}_j$ is the union of all regular expressions of the form $s_0 \Sigma^* s_1 \Sigma^* \dots \Sigma^* s_{n_j}$ where $\{s_0, \dots, s_{n_j}\}$ runs over the permutations $\Pi(F_j)$ of F_j . If some of the s_{ji} over-

lap (e.g., $s_{j0} = s_0s$ and $s_{j1} = ss_1$), the expression is more complicated because contracted terms of the form s_0s_1 have to be considered as well.⁷

Despite their complicated structure, strict regular expressions like (23) have an important application: they are useful to query set-valued attributes. In many corpora, some of the attributes are set-valued. Among them are grammatical (morphosyntactic) features, part of speech (if ambiguity is coded), and various meta-data features like author, genre and other domain classifications and so on.

Set-valued attributes can be conveniently coded as concatenation of the value strings, with a separator (e.g., a *space* character) inserted between the values and at both ends. The set $pos = \{\text{'noun'}, \text{'fem'}, \text{'pl'}\}$ for instance could be coded as (24).

(24) `'␣noun␣fem␣pl␣'`

In a corpus system that relies solely on existing string-based packages for regular expression matching, a complex and slow regular expression (e.g., (25)) would have to be constructed to find all values that contained, e.g., 'noun' and 'pl'.

(25) `␣noun(␣Σ*)?␣pl␣|␣pl(␣Σ*)?␣noun␣`

The exponential complexity introduced by the permutations can be avoided if the set values are ordered alphabetically or otherwise consistently.

When using the suffix-array based filtering approach, special query syntax can be devised to specify the filter disjuncts F_j directly; thus, the construction and parsing of complex regular expressions can be avoided, and high efficiency is guaranteed, in particular because the filters are strict by definition and no post-filtering with regular expressions has to be done.

Multi-valued attributes could in principle be treated in a similar way. (There is only a conceptual, but no technical difference between multi-valued and set-valued attributes.) I have chosen a different approach, in order to be able to handle the combination of multi-valued and set-valued attributes: attributes that may have multiple sets as values. Such a combination naturally arises when a morpho-syntactic annotation is not fully disambiguated: then, a word may be annotated with several feature sets, each corresponding to a different reading of the word.

In my approach, for multi-valued attributes, an additional attribute lexicon is maintained whose values are the separate values of the attribute (the values of the primary attribute lexicon being the concatenations of all the values in a corpus position, for each corpus position).

[7] See e.g. (24), where the strings may overlap in the separator character.

[4] REGEXPS OVER CORPUS POSITIONS: CUTTING THE EDGES

Corpus Workbench was the first corpus query system that used the full regular expression calculus over corpus positions and attribute values as query language. In CQP, the query language of Corpus Workbench, this calculus is further enhanced by a “labels” facility: labels (variables) can be attached to match positions in the query, and relational constraints (e.g. equality of attribute values) can be imposed on labeled positions. This makes CQP a versatile and powerful query language which has been adopted by several other systems.⁸ The strengths of CQP have helped to make Corpus Workbench one of the most popular corpus engines.

The query language I have implemented is a variant of CQP; I have adopted most of the syntax and semantics of CQP.

Formal characterization

Formally, the regular part of the query language can be characterized as a regular expression calculus over the alphabet Φ of constraints φ on corpus positions $\mathcal{C}$. Each constraint φ is a function

$$(26) \quad \varphi : \mathcal{C} \rightarrow \{ \text{true}, \text{false} \}$$

mapping a corpus position to *true* if the constraint is satisfied and to *false* otherwise.

The alphabet Φ is not a priori given. I choose, as CQP does, as a minimal alphabet Φ the set of constraints that are expressible as boolean combinations of regular expressions over attribute values. This minimal alphabet can be extended by other types of constraints, where in principle arbitrary functions $\varphi : \mathcal{C} \rightarrow \{ \text{true}, \text{false} \}$ could be used. A sensible constraint might use characteristics of the corpus like frequency counts and other types of statistics or external grammatical data. Corpus Workbench has a mechanism that allows the user to extend the alphabet by constraints written in the C programming language.

Evaluation

Using standard techniques, a regular expression R can be compiled into a minimal deterministic finite state automaton⁹ F over the alphabet Φ . The edges e of the automaton are labeled with constraints $\varphi \in \Phi$. Corpus positions $(c, c+1, \dots, c+k)$ are a match of F if there is a path $(e_0, \dots, e_k)$ through F starting in the initial state s_0 and ending in a final state of F such that $\varphi_i(c+i) = \text{true}$ for $i = 0, 1, \dots, k$.

Although F is deterministic over Φ , it is not deterministic in a procedural sense: If the labels φ_1 and φ_2 (where $\varphi_1 \neq \varphi_2$, since F is deterministic) of two edges e_1 and e_2 emerging from the same state s both evaluate to *true* in a given

[8] One of those is Manatee. TIGERSearch, the querying language of the TIGER treebanking system, is also heavily influenced by CQP, although it does not use finite state calculus beyond the string level.

[9] I use the terms automaton and network synonymously.

corpus position c , the paths along both e_1 and e_2 have to be explored further in the evaluation of the network.

Thus, when a starting position $c \in \mathcal{C}$ is given, a match $(c, c + 1, \dots, c + k)$ can be found by traversing the network depth-first, and backtracking when a state is reached whose outgoing edge labels all evaluate to *false* in the reached corpus position. Backtracking may also be used to find additional matches starting in c . A backtracking algorithm has the advantage that it is more space- and time-efficient than an algorithm which explores all paths in parallel, stepping from one corpus position to the next and keeping a list of all active paths. If the network is linear, that is, if each state has only one outgoing edge, there will be at most one match in each corpus position. Then, a backtracking algorithm is clearly the best choice. But when there is potentially more than one match, one is often interested in finding the shortest match only. Since a backtracking algorithm is not guaranteed to find the shortest match first, the shortest match can only be found after the full solution space has been searched. (In theory, the solution space can be arbitrarily large when the network contains cycles, but in practice, one limits the size of each match to, say, 100 positions.) Here, a parallel algorithm is preferable because it always finds the shortest solution first.¹⁰

Indexed evaluation, cutting edges

It is clearly not feasible to evaluate the finite state network in every corpus position. More preferable would be a way to utilize the inverted lexicon index to limit the evaluation to positions that in fact do match the edge constraints φ_i . It is not necessary to compute all matching positions for all constraints beforehand; it suffices to compute matching positions for a minimal set M of edges. A minimal set M is characterized by condition (27).

- (27) A set $M = (e_0, \dots, e_k)$ of edges of F is minimal if each path from the initial state to a final state goes through exactly one $e_i \in M$.

I call a minimal set of edges a set of *cutting edges* because they have the property that they cut the finite state network into two halves when the minimal edges are removed: one half that is connected to the initial state, and one half that is connected to the final state(s).

To a set $M = (e_0, \dots, e_k)$ of cutting edges we associate a set of partial networks $(F(e_0), \dots, F(e_k))$, where $F(e_i)$ consists of all paths of F that pass through e_i .

A partial network $F(e)$ can be split up into two networks $F^r(e)$ and $F^l(e)$, where $F^r(e)$ is the right partial network consisting of all paths starting in e , and $F^l(e)$ is the left partial network consisting of all paths ending in e , but with the direction of all edges reversed, such that the initial state of F becomes the final

[10] But see the discussion further down.

state of $F^l(e)$. The left vertex of e becomes the initial state of $F^r(e)$, and the right vertex becomes the initial state of $F^l(e)$.

In order to find the matches of F , first the corpus positions matching the constraint φ_i are calculated for each $e \in M$ using the inverted lexicon indices. Then, for each such corpus position c , a match $M^r(e) = (c, c + 1, \dots, c + m_r)$ of $F^r(e)$ is calculated. If a match is found, a corresponding match $M^l(e) = (c - m_l, \dots, c - 1, c)$ for $F^l(e)$ is calculated, but this time, the corpus is traversed in descending order. If also $M^l(e)$ exists, the partial matches $M^r(e)$ and $M^l(e)$ can be combined to a complete match $M(e) = M^l(e) \cup M^r(e) = (c - m_l, \dots, c - 1, c, c + 1, \dots, c + m_r)$ of $F(e)$ and of the original network F .

If there are several matches $M_j^r(e), j = 1, \dots, n$ of $F^r(e)$ and $M_k^l(e), k = 1, \dots, m$ of $F^l(e)$ starting in c , the set of matches of $F(e)$ is computed from the cross product of the sets of partial matches of $F^r(e)$ and of $F^l(e)$: Each $M_k^l(e) \cup M_j^r(e), k = 1, \dots, m, j = 1, \dots, n$ is a match of $F(e)$.

If one partial network ($F^r(e)$ or $F^l(e)$) is degenerate (i.e. consists of the edge e alone), the matches of $F(e)$ are precisely the matches of the other partial network ($F^l(e)$ or $F^r(e)$).

Optimal cutting edges

In the algorithms implemented in Corpus Workbench and in Manatee, the (cutting) edges for which index lookup is used are always the outgoing edges of the initial state. For each outgoing edge, all corpus positions are computed that match the corresponding constraint, and the partial network that starts in this edge is evaluated in all those positions.

This is a straightforward strategy, but it is not always the most efficient one.¹¹ It would be preferable to compute matching positions for an *optimal* set of cutting edges $\bar{M}$ as defined in (28):

- (28) $\bar{M}$ is optimal among all sets of cutting edges M if the sum of the number of matching corpus positions for all associated constraints $\varphi_0, \dots, \varphi_k$ is minimal.

When an optimal set $\bar{M}$ is chosen, the number of times partial networks have to be evaluated is minimal.

An optimal set $\bar{M}$ for F can easily be computed with the following algorithm:

- Compute the set of acyclic paths $\mathcal{P}$ of F . Each acyclic path $P \in \mathcal{P}$ can be represented as a sequence of distinct edges: $P = (e_0, \dots, e_k)$.
- Choose an edge e from the first path.

[11] See Evert & Hardie (2011) for a critical discussion of the implementation in Corpus Workbench.

- Continue recursively by choosing an edge from the next path in $\mathcal{P}$ that does not lie on any one of the already chosen paths. If such an edge does not exist, skip that path.
- Each set of possible choices gives an new set of edges M , and it is easy to see that for given M , the union of all the acyclic paths that contain one of the edges in M is the whole set of acyclic paths, and no two edges lie on the same path; thus M is a set of cutting edges. In addition, the sketched algorithm generates all possible sets of cutting edges.
- Among all M , an optimal set of cutting edges $\bar{M}$ can be determined using condition (28).

Example

As an example we consider the network F in (30). It is the minimal deterministic network associated to the regular expression (29),

$$(29) \quad cq^*(d|fp^*b)|ap^*b$$

where the letters a, b etc. stand for constraints on corpus positions. The initial state of F is labeled 0 and the only final state is 3. We assume that the position counts for those constraints are $|a| = 5, |c| = 50, |f| = 5, |d| = 2$ and $|b| = 35$. The position counts for p and q are irrelevant. It is easy to see that the set (a, d, f) is an optimal set of cutting edges for F . (The only other sets of cutting edges are (a, c) and (b, d) .)

(30)

The corresponding partial networks $F(f)$, $F(a)$ and $F(d)$ are shown in (31) and (32).

Matching strategies

A corpus query can be evaluated conforming to one of several matching strategies.

The most important matching strategy in practice is the strategy of shortest matches:

- (33) A *shortest match* M of a query F is a match that does not contain a shorter match.

One should note that evaluating a query by starting in matching positions of cutting edge constraints and combining shortest partial matches does not always result in shortest matches according to (33). To ensure that all found matches are shortest matches, the matches have to be tested for inclusion, and matches containing shorter matches have to be discarded. A simple example that illustrates in which way the cutting-edge algorithm for finding shortest matches can fail is given by the query (34) applied to the corpus (35).

- (34) $a\Sigma^*b$

- (35) $c a c c c c a c c b b b b$

Since $|a| = 2$ and $|b| = 4$, a is the optimal cutting edge of (34), and (34) is evaluated in corpus positions 2 and 7. The shortest matches of the corresponding automaton are the sequences ‘a c c c a c c b’ and ‘a c c b’, where the first one contains the second. On the other hand, no shortest matches are missed by the sketched algorithm either, because each shortest match must contain a position that is a match of a cutting edge constraint, thus, the partial networks are evaluated in that position and yield the shortest match.

Shortest matches and backtracking

A backtracking algorithm for evaluating a finite state network F with a shortest match strategy can be faster and more space efficient than a parallel algorithm if it can be guaranteed that the shortest match is found first. Therefore, it is important to decide whether the outgoing edges of each state in a given network can be ordered such that the depth first search that implements the backtracking algorithm does find the shortest match first, and to find an algorithm that sorts the edges in such a way.

In order to be able to formulate a sufficient condition and an ordering algorithm, I first define a *procedurally deterministic prefix* of a deterministic finite state automaton F over Φ , the alphabet of constraints φ on corpus positions $\mathcal{C}$.

- (36) A procedurally deterministic (p-deterministic) state d of F is a state whose outgoing edges e_i have the property that in any corpus position $c \in \mathcal{C}$, at most one of the edge constraints φ_i matches c .

A p-deterministic prefix of F is a dense¹² subnetwork F_D of F containing the start state of F such that every state of F_D is p-deterministic.

In other words, a p-deterministic network F_D is deterministic in every corpus position. The union of all p-deterministic prefixes of F is the maximal p-deterministic prefix $D(F)$ of F .

When the network F is evaluated in a corpus position c , there is for given $k > 0$ at most one path $(d_0, \dots, d_k)$ in F_D that matches the corpus positions $(c, c + 1, \dots, c + k)$. In consequence, among the partial networks F_d that start in the final states d of D , at most one is traversed in the traversal of F even when backtracking is necessary. This means that the edges of a p-deterministic prefix do not need to be ordered and that we can regard the subnetworks F_d in isolation when trying to order edges to guarantee that shortest matches are found first in a depth-first traversal.

[12] A subnetwork F' of F is *dense* if each state d of F' contains either all or none of the outgoing edges of d in F .

Ordering algorithm

Given a subnetwork F_d , we can try to order the edges e of the states s of F_d in the following way:

- (37) • Annotate each edge e of F_d by an interval $[x, y]$ where x is the length of the shortest path from the target state of e to a final state of F and y is the length of the longest such path, or ∞ if the length is unbounded.
- Order the outgoing edges of each state such that the intervals $[x_i, y_i]$ and $[x_j, y_j]$ of consecutive ordered edges e_i and e_j satisfy the condition $y_i \leq x_j$.

If not all outgoing edges of all states can be ordered in this way, a depth-first backtracking algorithm cannot be guaranteed to always find shortest matches first. It is easy to see that on the other hand, if all edges of all F_d can be consistently ordered, a depth-first backtracking algorithm that traverses outgoing edges in their order will always find shortest matches first: Assume that there are two matches $M^1 = (c, c+1, \dots, c+k)$ and $M^2 = (c, c+1, \dots, c+k, \dots, c+k+l)$ of F in c (with $l > 0$), with corresponding paths $E^1 = (e_0^1, \dots, e_k^1)$ and $E^2 = (e_0^2, \dots, e_{k+l}^2)$. Then $E^1 \cap F_D$ and $E^2 \cap F_D$ are (possibly empty) subpaths of the p-deterministic prefix F_D . By definition, they must coincide, and the two remaining path segments both lie in the same F_d , where d is the end point of the subpaths. Thus, they can be written as $E_d^1 = E^1 \cap F_d$ and $E_d^2 = E^2 \cap F_d$. Now, there are two possibilities: either E^1 is a subpath of E^2 , in which case M^1 is clearly found before M^2 , or E^1 and E^2 have a maximal (possibly empty) common prefix. Let e^1 and e^2 be the edges of E^1 and E^2 immediately following the common prefix, with intervals $[x^1, y^1]$ and $[x^2, y^2]$, and let l^1 and l^2 be the lengths of the path suffixes starting in e^1 and e^2 . By definition of the intervals, we have $l^1 \in [x^1, y^1]$ and $l^2 \in [x^2, y^2]$. Since $l^1 < l^2$, it follows that $y^1 \leq x^2$, and e^1 is traversed before e^2 . Hence, M^1 is found before M^2 .

When deciding whether a backtracking algorithm can be used to evaluate a given automaton F , it would clearly be best to start with a maximal p-deterministic prefix $D(F)$ of F . This would maximize the chance that the subnetworks F_d can be ordered consistently. It is however in many cases not feasible to calculate the maximal p-deterministic prefix. Depending on the constraints φ_i of the outgoing edges of a state s , determining that s is p-deterministic might amount to evaluating the constraints in every corpus position. I call the constraints φ_i of the edges of s *exclusive* if it can be decided that s is p-deterministic without knowledge of the corpus. Examples of exclusive constraints are constraints that match different literal values of an attribute, or regular expressions that can be shown to be exclusive. In the Corpuscle system, only exclusiveness tests for literal value

and certain regular expression constraints are used to calculate a p-deterministic prefix F_D .

CONCLUSION

I have presented two novel algorithms for efficient search in annotated corpora, used in the Corpuscle corpus engine. These algorithms are applied to static index files which are loaded into memory on-demand. Therefore, the design of the Corpuscle corpus engine can be seen as an evolutionary improvement over the design of e.g. Corpus Workbench, it does not radically depart from it.

There are also Corpus tools that use very different indexing and query evaluation techniques. An example is BlackLab,¹³ which builds on the document management system Apache Lucene. One advantage of using Lucene is that it offers incremental addition or deletion of texts; in contrast to Corpuscle and Corpus Workbench, the full Lucene/Blacklab index does not have to be generated anew when a document is added to it, which makes adding documents a much faster operation. Also systems that run on top of a relational database (e.g., the corpora at English-Corpora.org¹⁴) have the advantage of easy extensibility. The disadvantage of building on systems that are not specifically designed with corpus search in mind is that it can be challenging to translate complex corpus queries into efficient query expressions of the underlying system.

In addition to an expressive query language and fast query execution, many other factors are important in the design of a corpus engine. Among those are: a clean API that makes building front-end applications easy; support for a variety of result formats such as kwic concordances, tabular result aggregations, collocations, cooccurrence statistics and so on; query refinement/subqueries; searchable custom annotations; document styling (e.g. via XSL and CSS), and more. Most of these features have been implemented in the Corpuscle corpus engine and Web user interface, and Corpuscle has proven to be a very useful and versatile tool for a large variety of corpora.

ACKNOWLEDGMENTS

The work on Corpuscle and INESS-Search has been funded by the Meltzer foundation and the Norwegian Research Council.

DEDICATION

I dedicate this paper to the memory of Janne. Janne was for many years a valued colleague and project partner, whom I got to know as an open, kind and dynamic person full of spirit and character. We will all miss her.

[13] See <http://inl.github.io/BlackLab>

[14] See <http://www.english-corpora.org>

REFERENCES

- Baeza-Yates, Ricardo A. & Gaston H. Gonnet. 1996. Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM* 43(6), 915–936.
- Evert, Stefan & Andrew Hardie. 2011. Twenty-first century corpus workbench: Updating a query architecture for the new millennium. In *Proceedings of the corpus linguistics 2011 conference*. University of Birmingham.
- Manber, Udi & Gene Myers. 1993. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing* 22(5), 935–948.
- Meurer, Paul. 2012a. Corpuscle – a new corpus management platform for annotated corpora. In *Exploring newspaper language*, edited by Gisle Andersen, 31–49. John Benjamins Publishing Company.
- Meurer, Paul. 2012b. INESS-search: A search system for LFG (and other) treebanks. In *Proceedings of LFG'12 conference*, edited by Miriam Butt and Tracy Holloway King, 404–412. CSLI Publications.

CONTACT

Paul Meurer
University of Bergen
paul.meurer@uib.no