Matrix and Double-Array Representations for Efficient Finite State Tokenization

Nils Diewald
Leibniz Institute for the German Language
Mannheim, Germany
diewald@ids-mannheim.de

Abstract

This paper presents an algorithm and an implementation for efficient tokenization of texts of space-delimited languages based on a deterministic finite state automaton. Two representations of the underlying data structure are presented and a model implementation for German is compared with state-of-the-art approaches. The presented solution is faster than other tools while maintaining comparable quality.

Keywords: Tokenization, Finite State, Corpora

1. Introduction

Tokenization, i.e. the segmentation of a text string into “distinct meaningful units” (Kaplan, 2005) is a fundamental step in the preparation of linguistic corpora. Character sequences are subdivided (like “Look␣it␣up ␣at␣p.␣124!␣;-)” into “Look|it|up|at|p.|124|!|;-)|”) to make the individual units accessible for search engines and further linguistic analysis. Since errors in tokenization often have a significant impact on further processing and analyses, high accuracy is of great importance. As ambiguities concerning sentence boundaries have to be resolved for tokenization, they are usually marked in the same step.

Although tokenization – especially for space-delimited languages such as English or German – is considered one of the simpler applications of natural language processing (NLP) and is often regarded as a solved problem, there are some cases where programmatic recognition of token boundaries pose challenges and naïve approaches may fail, for example, in distinguishing the period character at the end of an abbreviation from marking the end of a sentence. More recent phenomena of computer-mediated communication (CMC), such as emoticons, URLs, or email addresses, pose difficulties in particular.

Tokenization is rarely a time-critical process, especially in preprocessing for much more time-consuming syntactic or semantic analyses. And the quality of the results is clearly the most important measure for evaluating this task. However, in the case of very large corpora in research data preparation, tokenization can be challenging – and speed of processing, accompanied by low resource consumption, can be an important criterion in deciding which tool to choose.

In many areas of NLP, rule-based approaches have been replaced by machine-learning (ML) methods in recent years. This is due to more efficient algorithms and better hardware for the implementation of such solutions on the one hand, and to the availability of large annotated corpora for training these systems on the other hand. Tokenization and sentence segmentation are still exceptions to this (although there are significant differences with respect to different languages). The main reason is that accuracy of rule-based tokenizers for space-delimited languages is already very high. For German, for example, rule-based approaches continue to outperform ML approaches significantly both in terms of accuracy and speed (Ortmann et al., 2019; Diewald et al., 2022).

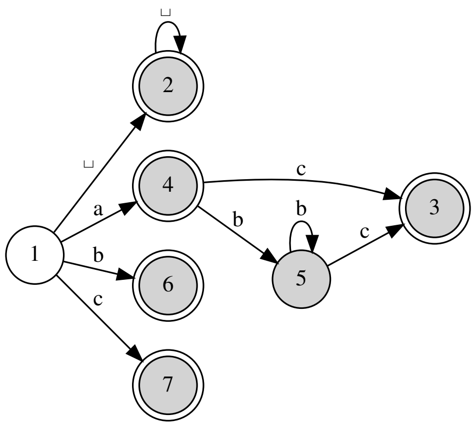

Figure 1: Lexical analyzer for tokens `a`, `b`, `c`, `ab*c` and whitespace sequences (␣$^+$).

1.1. Lexical Analyzers

Rule-based tokenizers and sentence segmenters have traditionally been based on *lexical analyzers* (Aho et al., 2007, ch. 3) using a general purpose lexical scanner generator such as Lex (Lesk and Schmidt, 1975) or modern successor systems like Flex, JFlex or Ragel. Rules for lexical units are formulated as regular expressions and transformed into a deterministic finite state automaton (FSA), which linearly searches the input stream, executes arbitrary code when reaching terminal states, and for ambiguous inputs follows the principle

Proceedings of the CMLC-10 Workshop (@LREC 2022), pages 20–26
Marseille, 20 June 2022

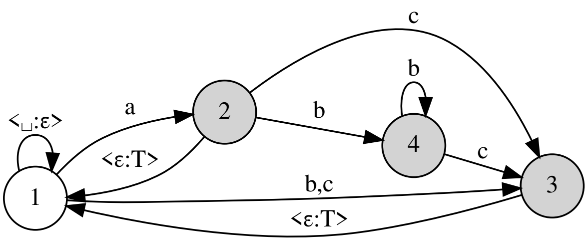

Figure 2: Tokenizing automaton segmenting `a`, `b`, `c`, `ab*c`, ignoring whitespace sequences (␣^+) and introducing token boundaries (T).

of the longest match (see Fig. 1).

Modern rule-based tokenizers also follow this approach, for example the Stanford Tokenizer[1], BlingFire[2], or KorAP-Tokenizer[3] (Kupietz and Diewald, 2020). Tokenizers that rely on dictionaries to vectorize an input stream follow a similar approach (Song et al., 2021).

1.2. Finite State Transducers

An alternative – or generalization – of this approach is the tokenization using finite state transducers (FST; Beesley and Karttunen, 2003, ch. 9.2; Beesley, 2004). FSTs are finite state automata with translating edges. They not only accept symbol sequences of an input string, but return for each input symbol an output symbol and thus generate for each accepted input string at least one output string. By supporting empty characters (ε) in input and output, i.e. symbols which do not consume or produce any characters, it is possible to formulate a transducer that converts an input stream into an arbitrarily segmented output stream (see Fig. 2).

Kaplan (2005) describes an algorithm based on an FST representation of a tokenizer. Following a breadth-first traversal, an incremental composition operation is performed on the tokenizing FST with a linear text FSA. The output of the operation is an FSA of all possible tokenizations (or a sequence of these FSAs), with the ambiguities still intact to be resolved by higher-level lexical constraints.

1.3. Further Models

Further approaches of rule-based tokenizers extend these models, for example, to a list of finite state automata that are applied in a defined order (Proisl and Uhrig, 2016), or by applying context-free rules recursively (Graën et al., 2018, or SpaCy[4]).

[1]`http://nlp.stanford.edu/software/tokenizer.shtml`
[2]`https://github.com/Microsoft/BlingFire`
[3]`https://github.com/KorAP/KorAP-Tokenizer`
[4]`https://spacy.io/usage/linguistic-features#tokenization`

2. Data Structure

While Lex-like scanner generators allow arbitrary code executions at terminal nodes, and FSTs support arbitrary character transitions, for a finite state tokenizer the transition types can be reduced to three cases:

Identity: The input symbol corresponds to the output symbol (e.g., a character within a word);

Deletion: The input symbol can be ignored (e.g., a whitespace character between word boundaries);

Token Boundary: The input symbol is followed by the end of a token (e.g., a dot at the end of an abbreviation).

Beesley (2004) proposes a mechanism for formulating an FST-based tokenizer, which inserts a transition following every acceptable token, which consumes an empty character (i.e., can always be traversed) and produces a token boundary marker (T). The above rules can then be mapped to three types of edges in the automaton (see Fig. 2 for an application of these rules):

$<?>$	for the identical output of arbitrary input symbols;
$<? : \varepsilon>$	for the deletion of arbitrary input symbols;
$<\varepsilon : T>$	for marking token boundaries without consuming an input symbol.

Compared to an FSA or FST, terminal nodes do not play a role in finite state tokenizers – the set of terminal nodes is empty. We can represent it accordingly as a quintuple:

Σ	Finite alphabet of the input language ($\varepsilon \in \Sigma$);
Φ	Finite set of states;
δ	State transition function;
s_1	Initial state;
δ_D	Finite set of all $<? : \varepsilon>$ transitions.

Reducing the transducer to these simple rules guarantees, that for an input symbol to be consumed exactly one output symbol exists. Ambiguity with respect to token boundaries arises only when traversing $<\varepsilon : T>$.

2.1. Matrix Representation

A standard representation of all transitions of a finite state automaton is a state transition table (Tab. 1 shows the matrix representation of the automaton in Fig. 1). Additional information includes the initial state and terminal nodes.

A transducer would encode output symbols in addition to the destination node in this table. Due to the reduced transition types of the tokenizer, this can be simplified by encoding all identity transitions with a positive sign and all deletion transitions with a negative sign[5] (see

[5]In an implementation, the most-significant bit could be used for marking.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7
a	4	0	0	0	0	0	0
b	6	0	0	5	5	0	0
c	7	0	0	3	3	0	0
␣	2	2	0	0	0	0	0

Table 1: State Transition Table for FSA of Fig. 1

	s_1	s_2	s_3	s_4
a	2	0	0	0
b	3	4	0	4
c	3	3	0	3
␣	-1	0	0	0
ε	0	1	1	0

Table 2: State Transition Table for the reduced FST of Fig. 2

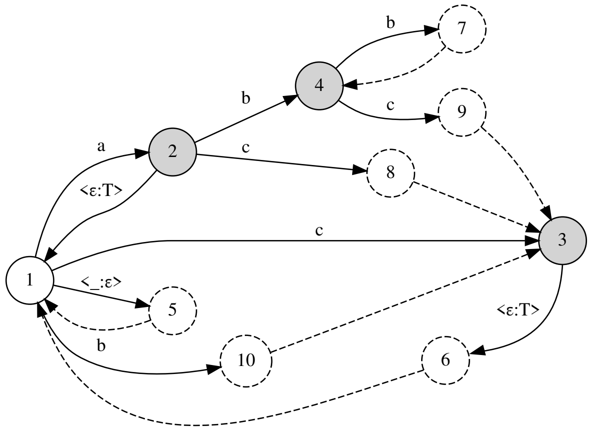

Figure 3: Double-array FST resembling the automaton in Fig. 2

Tab. 2, esp. $\delta(s_1, ␣) = -1$ for an example of a deletion transition). Since ε transitions by definition only mark token boundaries (T) no additional encoding is necessary.

2.2. Double-Array Representation

However, the matrix representation can cause a problem: Not only the number of states in the automaton has an influence on the model size and thus on the required storage space, but also the size of the alphabet $|\Sigma|$. This can be an issue depending on the language to model and the sparseness of the transition table. Alternatively, the finite state tokenizer can be implemented based on a double-array (DA) trie (Aoe, 1989) as a DA finite state machine (Mizobuchi et al., 2000). In a DA trie the state transition function of an automaton can be represented in two one-dimensional numeric arrays of equal length (`base` and `check`). Both state and input symbols are encoded as numeric values > 0. A state transition $t_0 = \delta(t, x)$ is thereby valid if:

$$t_0 = \texttt{base[}t\texttt{]} + \texttt{code[}x\texttt{]}$$
$$\texttt{check[}t_0\texttt{]} = t$$

A target state is recorded in the `base` array at the position of the sum of the current state and the numeric code of the input symbol. In the construction of the DA trie[6] care is taken, that the transitions are stored compactly and possibly overlapping, therefore in the `check` array at the target position the parent state must be checked.

The difference between a trie and a regular FSA is that the in-degree of a state in the FSA can be > 1 and that circular structures may exist. While the representation as a DA allows for circular structures, it can not represent nodes with an in-degree > 1. Mizobuchi et al. (2000) therefore introduce groups of "separate states" for nodes that have an in-degree > 1, pointing to a "representative state" to encode FSAs in DA structures (Fig. 3 shows the automaton from Fig. 2 with separate states in dashed circles pointing to representative states). To model the relationship of separate states to representative states in the DA, they introduce an intermediate step in the `base` array, which encodes with a negative sign. If `base[`t`]` has a negative sign, the transition corresponds to a separate state whose value points to the representative state. Accordingly, in addition to the condition above, the following is true:

$$t_t = \texttt{base[}t\texttt{]} + \texttt{code[}x\texttt{]}$$
$$t_0 = \begin{Bmatrix} \texttt{base[}|t_t|\texttt{]},\ if\ t_t < 0 \\ t_t,\ otherwise \end{Bmatrix}$$

When traversing the edges, this intermediate step must be taken into account.

Corresponding to this mechanism, $<?:\varepsilon>$ edges can be represented in the double array to model a finite state tokenizer, in that for transitions with the destination t the value in `check[`t_0`]` is given a negative sign. Note, that this check must be performed before the resolution of a separate state (Tab. 3 shows one possible representation of the automaton in Fig. 3 as an extended DA FSA with representative state references and deleting transitions).

As this representation is independent of $|\Sigma|$, it can lead to smaller models under certain conditions.

3. Algorithm

Algorithm 1 shows the simplified (see below) tokenization of an input sequence in into the tokenized output sequence out. The algorithm is representation-agnostic, the only difference to be noted is that with a DA representation, the sign of the target node comes from the `check` and the value corresponds to the representative state from `base`.

Valid transitions: For each input character in_i the transition $\delta(t, in_i)$ is checked in the automaton. Characters leading to targets with a positive sign are written

[6] Regarding the efficient construction of static DA tries, please refer to Niu et al. (2013).

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
`base`	1	4	-3	3	-1	8	-3	-1	-1	-6
`check`	10	1	1	1	-1	2	2	4	2	6

Table 3: Extended double-array FSA of Fig. 3, with `code[`a`]`=1, `code[`b`]`=2, `code[`c`]`=3, `code[␣]`=4, `code[`ε`]`=5. The length of the DA is stored in `check[1]`.

Algorithm 1: Main tokenization loop

Input: in is a character stream

Output: out is a tokenized character stream

1 $newChar \leftarrow true$;
2 $i \leftarrow 0$; `// position in` in
3 $j \leftarrow 0$; `// position in` out
4 $t_\varepsilon \leftarrow 0\ (\neq s_1)$; $i_\varepsilon \leftarrow 0$; $j_\varepsilon \leftarrow 0$; $t \leftarrow s_1$;
5 **while** $i < |in|$ **do**
6   **if** $newChar$ **then**
7     $char \leftarrow in_i$;
8   **if** $char \notin \Sigma$ **then** $char \leftarrow ?$;
9     $t_0 \leftarrow t$;
10   **if** $\delta(t_0, \varepsilon) \neq 0$ **then** $t_\varepsilon \leftarrow t_0$; $i_\varepsilon \leftarrow i$; $j_\varepsilon \leftarrow j$;
11   $t \leftarrow \delta(t_0, char)$;
12   **if** $t = 0$ **then**
13   **if** $char \neq \varepsilon$ **and** $t_\varepsilon \neq 0$ **then**
14       $t_0 \leftarrow t_\varepsilon$; $i \leftarrow i_\varepsilon$; $j \leftarrow j_\varepsilon$; $char \leftarrow \varepsilon$;
15   **else**
16   **if** $t_o = s_1$ **then**
17         $i \leftarrow i + 1$; $out_j \leftarrow in_i$;
18   **else**
19         $t_0 \leftarrow s_1$;
20       $out_j \leftarrow T$; $j \leftarrow j + 1$; $newChar \leftarrow true$;
21   **restartLoop**
22     $newChar \leftarrow false$
23   **restartLoop**
24   **if** $char = \varepsilon$ **then**
25     $out_j \leftarrow T$; $j \leftarrow j + 1$;
26   **else**
27     $i \leftarrow i + 1$;
28   **if** $t > 0$ **then**
29       $out_j \leftarrow char$; $j \leftarrow j + 1$; $newChar \leftarrow true$;
30   **if** $t < 0$ **then** $t \leftarrow -t$;
31   $newChar \leftarrow true$;

unchanged to the output stream (see line 28f). Characters that lead to targets with a negative sign are consumed only.

Backtracking: $<\varepsilon : T>$ edges allow a transition in the automaton without consuming a character of the input stream. This means that whenever a transition $\delta(t, in_i)$ is available, a possible transition $\delta(t, \varepsilon)$ must also be considered (cf. δ(2,b) in Fig. 2). Since this by definition implies a token boundary mark T, this can be used for backtracking semantics to follow a longest-match strategy (Lesk and Schmidt, 1975): During traversion, the last available $<\varepsilon : T>$ transition is remembered (see line 10), but character consumption is always prioritized. If a character cannot be consumed during traversion, the system repositions out and in, jumps to the last $<\varepsilon : T>$ source state (see line 13–14), traverses it, and continues.

Invalid transitions: If no valid transition of an input symbol exists and backtracking is not possible, a token boundary marker T is added to the output stream and the remaining input stream is continued from the initial state s_1 of the tokenizer. If the automaton is already initial, a character is consumed beforehand (see lines 15–21). This guarantees robust output of all input data with all automata. In carefully designed tokenizers, this behavior is rarely triggered.

The representation of the algorithm is simplified in that an implementation (and also the model) must be able to handle characters $\notin \Sigma$. In addition, special treatments are necessary with respect to the end of the processing. By concatenating several token boundary markers T, it is also possible to mark sentence boundaries (see Sec. 4.2).

The worst time complexity of the algorithm is $O(nm)$, where $n = |in|$ and m is the maximum path length excluding $<\varepsilon : T>$ edges. Intermediate memory requirement corresponds to the length of the text, whereby the processing can be handled by a buffer which can be flushed after each successfully parsed token.

4. Implementation

4.1. Datok

Datok (Diewald, 2022) is an implementation of a finite state tokenizer based on the aforementioned algorithm and datastructures. It is written in Go as a command line tool and was designed to be compatible with KorAP-Tokenizer.

Datok relies on XFST (Beesley and Karttunen, 2003) for the construction of its automaton in the free implementation of Foma (Hulden, 2009) (see next section; other FST toolkits should be equally suitable).

To create an automaton that can be interpreted by Datok, first Foma must compile the rule set into a compatible FST and subsequently Datok must convert the FST into a finite state tokenizer (optionally in matrix or DA representation). The final automaton can then be applied to arbitrary data input streams, and can output different forms of tokenization data (like new-line delimited surface forms or character offset information).

4.2. Construction

While the implementation of the algorithm and the underlying data structures are relatively simple, the complexity lies in the automaton and thus the challenge in its construction.

Rule creation in XFST essentially follows Beesley (2004), with the supplement to restrict rule formulation to valid transitions $<?>$, $<?:\varepsilon>$, and $<\varepsilon:T>$. The special symbol "`@_TOKEN_BOUND_@`" is introduced as the token bound marker.

A very simple tokenizer that follows the introduced rules, can be seen in Listing 1.

```
1  define TB "@_TOKEN_BOUND_@";
2  define WS [" "|"\u000a"|"\u0009"];
3  define PUNCT ["."|"?"|"!"];
4  define Char \[WS|PUNCT];
5  define Word Char+;
6
7  ! Compose token boundaries
8  define Tokenizer
9      [[Word|PUNCT] @-> ... TB] .o.
10 ! Compose Whitespace ignorance
11     [WS+ @-> 0] .o.
12 ! Compose sentence ends
13     [[PUNCT+] @-> ... TB \/ TB _ ];
14 read regex Tokenizer;
```

Listing 1: Compliant Tokenizer written in XFST

First, the token inventory of the tokenizer is defined using regular expressions (lines 1–5). The *direct replacement* operator "`@->`" (Karttunen, 1996), which performs a replacement on the longest possible path, and the context operator "`...`", which allows to insert arbitrary symbols around a match, are helpful for the creation of $<\varepsilon:T>$ transitions. In the example tokenizer these operators append the token boundary marker to the longest possible matches of all entries of the token inventory (line 9).

The $<?:\varepsilon>$ transitions are realized by replacing arbitrary characters with ε ("0" in XFST; used in the example for whitespace characters in line 11).

By using the direct replacement rules it is also possible to specify sequences of token boundary markers which can be interpreted separately by an implementation. For example, it is possible to mark sentence boundaries within the framework (line 13).

The *direct replacement* operations yields an unambiguous transducer for the unique processing of input streams. Unfortunately, such automata (especially in intermediate steps during compilation) can reach a very large size and thus require an enormous amount of resources. Due to the longest-match and backtracking strategy of the algorithm, however, it is possible to achieve unique outputs even with ambiguous transducers. Thus, when constructing the finite state tokenizer in XFST, automata of individual token inventories can first be created separately using direct replacement operators and then be unified, e.g., for the composition of sentence ending rules and whitespace treatment. This flexible construction of the tokenizer enables a trade-off in terms of model size and processing speed (which decreases when backtracking is utilized to a great extent).

4.3. Benchmarks

In a real world tokenizer, these rules are more complex with respect to applicable contexts for token and sentence boundaries and the defined automata of the token inventory (e.g., abbreviation lists, emoticons, numbers). Datok (v0.1.5) contains a real world tokenizer for German with more than 18 thousand states, more than 2 million edges and $|\Sigma| = 167$. The ruleset is based on preliminary work by KorAP-Tokenizer and Çöltekin (2014). The matrix representation requires ~10.9 MB of memory, the DA representation ~18.5 MB (with a load factor[7] of ~70.8%).

Diewald et al. (2022) presents a detailed comparison of 15 different tools (both ML and rule based approaches) for the tokenization and sentence segmentation of German language data including Datok. Table 4 gives a summary of the results regarding the quality of Datok in the form of F_1 values with respect to tokenization and sentence segmentation in 3 different corpora: Version 2.9 of the German Universal Dependency GSD Corpus (McDonald et al., 2013) and the CMC and Web corpora of the EmpiriST Shared Task Challenge (Beißwenger et al., 2016). While all tested tools achieve values well above 99% for the tokenization of the UD-GSD corpus, the F_1 values for the CMC and Web corpora are comparatively very high.[8] The values for sentence segmentation are in the middle range.

	Tokens			Sentences
	UD-GSD	CMC	Web	UD-GSD
F_1	99.45%	98.79%	99.21%	97.60%

Table 4: Evaluation of the quality of Datok's sentence and token boundary detection for German (v0.1.5).

Figure 4 presents the performance in tokens per millisecond at different batch sizes (here logarithmically represented in $2^x \times 1000$ tokens) of four different tokenizers: Datok (in matrix and DA representation), BlingFire (as the fastest competitor tokenizer according to Diewald et al. 2022; v0.1.8 with the "wbd.bin" model using the Python API), KorAP-Tokenizer (v2.2.2), and Stanford Tokenizer (v4.4.0[9]; probably the most widely used tokenizer tool). The test

[7] I.e. the proportion of non-empty elements to all elements in the representation.

[8] For a detailed account of the evaluation, please refer to Diewald et al. (2022). The full evaluation suite including all results is available at `https://github.com/KorAP/Tokenizer-Evaluation`.

[9] Including sentence segmentation.

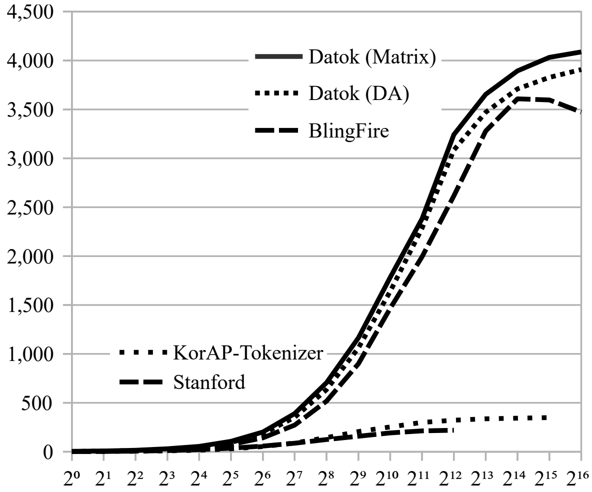

Figure 4: Benchmarks in t/ms for different batch sizes (averaged over 10 runs).

system is an Intel Xeon CPU E5-2630 v2 @ 2.60GHz with 12 cores and 64 GB of RAM. As can be seen, model loading and startup time has a big impact on very short texts but becomes negligible for longer texts.[10] Datok can process up to ∼4,000 t/ms in matrix representation, ∼3,900 t/ms in DA representation, and BlingFire ∼3,600 t/ms. KorAP-Tokenizer (∼350 t/ms) and Stanford Tokenizer (∼220 t/ms) are significantly slower.

The implementation as a DA is slower than the matrix implementation (presumably due to the additional parent check for each traversion and the resolution of separate states), but still competitive and therefore a possible variant for implementations with large alphabets.

In view of the processing of very large corpora, such speed differences can play a significant role. Datok (like KorAP-Tokenizer) was primarily developed for tokenizing the German reference corpus DeReKo (Kupietz et al., 2018), which currently comprises over 50 billion tokens. Complete processing of this corpus on the test system would take ∼13.5h using Datok (in matrix representation; assuming a batch size of 100,000 tokens and a single core), BlingFire ∼33h, KorAP-Tokenizer ∼8 days, and Stanford Tokenizer (including sentence segmentation) ∼12.5 days. For the same task, some other tools require several years to complete and can therefore be considered impractical in this application scenario (Diewald et al., 2022).

5. Summary and Outlook

The algorithm and the corresponding data structures presented in this paper show a high performance in tokenizing large corpora in the implementation of Datok. At the same time, the model allows complex rule sets that achieve a very high quality for space-delimited languages. Thus, Datok can be used as a suitable tool in research data preparation.

However, there are some limitations associated with the algorithm that need to be taken into account. For example, long-distance relationships between tokens (Graën et al., 2018) cannot be used for disambiguation (e.g., opening single quotes that can help distinguish a closing single quote from being used as an apostrophe). Also, the left longest-match rule prevents valid tokens from being further subdivided, even though this may result in shorter segments on the right side of the analysis (e.g., the string “Go␣tohttp://google.com/”, in which a space was omitted by mistake, would be tokenized using common word and URL rules into “Go|tohttp|:|/|/|google|.|com|/|” instead of “Go|to|http://google.com/|”). Since the output produced is unambiguous and no longer contains possible interpretations, ambiguities can not be resolved by higher-level lexical constraints (Kaplan, 2005).

Extensions to the algorithm and the data models are possible. Token boundaries could be marked to modify the backtracking behaviour (e.g., to exempt some ε edges from being considered as backtracking positions). And, specifically in matrix representation, token classes can be associated with token boundary markers (e.g., to additionally mark that a token is an URL), as is common in several tokenizer tools. This extension would also make it possible to resolve parts of the aforementioned restrictions by re-evaluating doubtful cases based on token classes in a second step.

Currently, Datok is in the evaluation phase for future use in tokenizing DeReKo, for which KorAP-Tokenizer is presently being used. Datok is open source[11] and published under the Apache 2.0 License. Language models for English and French are under preparation.

6. Bibliographical References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools*. Pearson Education, Addison-Wesley, second edition.

Aoe, J.-I. (1989). A fast digital search algorithm using a double-array structure. *Systems and Computers in Japan*, 20(7):92–103.

Beesley, K. R. and Karttunen, L. (2003). *Finite State Morphology*. CSLI Studies in Computational Linguistics. CSLI Publications.

Beesley, K. R. (2004). Tokenizing Transducers. Technical report, Xerox Research Centre Europe, October.

Beißwenger, M., Bartsch, S., Evert, S., and Würzner, K.-M. (2016). EmpiriST 2015: A Shared Task on the Automatic Linguistic Annotation of Computer-Mediated Communication and Web Corpora. In *Proceedings of the 10th Web as Corpus Workshop*, pages 44–56, Berlin, August. Association for Computational Linguistics.

[10] Caching effects cannot be ruled out, since batches are based on a concatenated, repetitive text of ∼98 thousand tokens.

[11] `https://github.com/KorAP/Datok`

Çöltekin, Ç. (2014). A set of open source tools for turkish natural language processing. In Nicoletta Calzolari, et al., editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 1079–1086, Reykjavik, Iceland. European Language Resources Association (ELRA).

Diewald, N., Kupietz, M., and Lüngen, H. (2022). Tokenizing on scale – Preprocessing large text corpora on the lexical and sentence level. In *Proceedings of EURALEX 2022*, Mannheim, Germany, July.

Diewald, N. (2022). Datok. Software; doi:10.5281/zenodo.6427259, `https://github.com/KorAP/Datok`.

Graën, J., Bertamini, M., and Volk, M. (2018). Cutter – a universal multilingual tokenizer. In Mark Cieliebak, et al., editors, *Swiss Text Analytics Conference*, number 2226 in CEUR Workshop Proceedings, pages 75–81. CEUR-WS, June.

Hulden, M. (2009). Foma: A finite-state toolkit and library. In *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*, pages 29–32.

Kaplan, R. M. (2005). A Method for Tokenizing Text. In Antti Arppe, et al., editors, *Inquiries into Words, Constraints and Contexts. Festschrift for Kimmo Koskenniemi on His 60th Birthday*, CSLI Studies in Computational Linguistics Online, pages 55–64. CSLI Publications, Ventura Hall.

Karttunen, L. (1996). Directed Replacement. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 108–115, Santa Cruz, California, USA, June. Association for Computational Linguistics.

Kupietz, M. and Diewald, N. (2020). KorAP-Tokenizer. Software. doi:10.5281/zenodo.5040449, `https://github.com/KorAP/KorAP-Tokenizer`.

Kupietz, M., Lüngen, H., Kamocki, P., and Witt, A. (2018). The German reference corpus DeReKo: New developments – new opportunities. In *Proceedings of the 11th International Conference on Language Resources and Evaluation (LREC 2018), 7-12 May 2018, Miyazaki, Japan*, pages 4354–4360, Paris, France, May. European language resources association (ELRA).

Lesk, M. E. and Schmidt, E. (1975). Lex - A Lexical Analyzer Generator. Technical Report 39, Bell Laboratories, Murray Hill, NJ.

McDonald, R., Nivre, J., Quirmbach-Brundage, Y., Goldberg, Y., Das, D., Ganchev, K., Hall, K., Petrov, S., Zhang, H., Täckström, O., Bedini, C., Bertomeu Castelló, N., and Lee, J. (2013). Universal Dependency Annotation for Multilingual Parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia, Bulgaria, August. Association for Computational Linguistics.

Mizobuchi, S., Sumitomo, T., Fuketa, M., and Aoe, J.-i. (2000). An efficient representation for implementing finite state machines based on the double-array. *Information Sciences*, 129(1):119–139, November.

Niu, S., Liu, Y., and Song, X. (2013). Speeding Up Double-Array Trie Construction for String Matching. In Yuyu Yuan, et al., editors, *Trustworthy Computing and Services*, Communications in Computer and Information Science, pages 572–579, Berlin, Heidelberg. Springer.

Ortmann, K., Roussel, A., and Dipper, S. (2019). Evaluating off-the-shelf NLP tools for german. In *Proceedings of the 15th Conference on Natural Language Processing (KONVENS 2019): Long Papers*, pages 212–222, Erlangen, Germany. German Society for Computational Linguistics & Language Technology.

Proisl, T. and Uhrig, P. (2016). SoMaJo: State-of-the-art tokenization for German web and social media texts. In *Proceedings of the 10th Web as Corpus Workshop*, pages 57–62, Berlin, August. Association for Computational Linguistics.

Song, X., Salcianu, A., Song, Y., Dopson, D., and Zhou, D. (2021). Fast WordPiece Tokenization. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2089–2103, Online and Punta Cana, Dominican Republic, November. Association for Computational Linguistics.