*© 2016 Institute for the German Language, Mannheim*

# The *ShortestMismatcher* tool

Peter Meyer, Institute for the German Language, Mannheim

## Introduction

This document provides information on the functionality, usage, and algorithmic foundations of an open source software tool which accompanies the paper titled "The statistical tradeoff between word order and word structure – large-scale evidence for the principle of least effort" by Alexander Koplenig, Peter Meyer, Sascha Wolfer, and Carolin Müller-Spitzer (currently under review). A preprint of the paper is available here: http://www.owid.de/plus/eebib2016/project.html.

## Functionality of the tool

The command line tool *ShortestMismatcher* creates, for any given UTF-8-encoded input plaintext file, a two-column tab-separated UTF-8 output CSV file; in this output file, the first CSV column lists the characters of the input file in order of appearance, whereas the second column indicates the **match-length** or **shortest mismatch number** for the positions corresponding to the character tokens. The shortest mismatch number of a text position is the length of the shortest substring starting at this position that is *not* also a substring contained in the part of the text before this position. To give a simple example, for an input file containing the text `bananas`, the output file looks like this:

```
b	1
a	1
n	1
a	3
n	3
a	2
s	1
```

In this example, the shortest mismatch number for the position of the second occurrence of the letter `n` is 3, which is one plus the longest substring starting at that position that is a repetition of a substring contained in the text before that position, viz. `na`.

The shortest mismatch numbers for a given text can be used for certain entropy estimators; see Kontoyiannis et al. (1998) for an overview.

## System requirements and program invocation

The *ShortestMismatcher* binary is a Java archive file, `shortestmismatcher.jar`, that can be executed without further dependencies. Only a standard Java Runtime Environment (JRE), version 8 or higher, must be installed on the system. To get a brief overview of the execution options, invoke

```
java -jar /path/to/shortestmismatcher.jar
```

In particular, the tool may be run in two different operation modes:

a) In **single file mode** a single file is processed: Invoke the program with the paths of the input and the output file as parameters:
```
java -XmxZZZZM -jar /path/to/shortestmismatcher.jar
/path/to/input/text/file /path/to/output/csv/file
```
The output file must not exist at the time of invocation.

b) In **directory mode** all files with the file name suffix `.txt` that reside in a user-specified input directory are processed; the file names of the resulting CSV files are built from the corresponding input file names by adding the suffix `.sm.txt` and are stored in a separate output directory. The tool must be invoked with three command line parameters:
```
java -XmxZZZZM -jar /path/to/shortestmismatcher.jar
/path/to/input/directory /path/to/output/directory N
```
Here, `N` must be replaced by the desired maximum number of threads for file processing, which must not be greater than the number of CPU cores available on the machine. At any time, a maximum of $N$ files will be processed in parallel; the number of `*.txt` files in the directory is not limited.

In all cases, the maximum available Java heap size should be specified explicitly – the memory requirements grow approximately linearly with the number of threads and the file size. The sequence `ZZZZ` in the above commands must be replaced by an integer number (not necessarily with four digits) indicating the maximum heap memory size, in megabytes, that should be available to the program (note that actual memory usage can be somewhat higher). Note that there must not be any whitespace within this parameter, so a correct invocation could, for example, look like this: `java -Xmx3500M …` Please note that paths may be enclosed in standard double quotes, which is useful, amongst other things, if there are whitespace characters within the path.

In directory mode, the available memory must be sufficient for all $N$ parallel processing threads.

Please note that the unicode null character `\0` must not appear in the input files. Other than that, any unicode character is allowed. The null character never appears in normal plaintext files anyway.

## License

The tool and its source code is distributed under the terms of the GNU General Public License as published by the Free Software Foundation. For details, see the source code.

## Overview of the algorithm used in the tool

*Preliminary Definitions*

In what follows, a **substring** of a string $S$ is a sequence of consecutive contiguous characters in $S$. We write $S[a..b]$ to denote the substring of $S$ beginning at position $a$ and running up to and including position $b$. All positions and indexes are taken to be 0-based in this exposition. A **prefix** of $S$ is a substring $S[0..a]$ that starts at the very first character token of $S$, and a **suffix** of $S$ is a substring $S[a..(\text{length}(S)-1)]$ that ends at the very last character token of $S$.

*Informal sketch of the algorithm*

All suffixes of the input text $T$ are sorted in lexicographical order (using an arbitrary ordering of the alphabet used). A data structure (technically, a **suffix array** with an accompanying **LCP array**) is constructed that can be represented as a table that lists, for each suffix $S$, its **rank** (position of $S$ in the lexicographical order), its **index** (position of the first character of $S$ in $T$) and the length of **longest common prefix** (henceforth, *LCP*) of $S$ and the suffix preceding $S$ w.r.t. rank in lexicographical order. Note that there is no such LCP for the suffix with rank 0. The suffix array of the Text $T$ = bananas can be represented as follows, given standard latin alphabet order:

| *suffix* | *rank* | *index* | *lcp(rank, rank-1)* |
|---|---|---|---|
| ananas | 0 | 1 | - |
| anas | 1 | 3 | 3 |
| as | 2 | 5 | 1 |
| bananas | 3 | 0 | 0 |
| nanas | 4 | 2 | 0 |
| nas | 5 | 4 | 2 |
| s | 6 | 6 | 0 |

In what follows, we will write $\mathrm{lcp}(i,j)$ for the length of the LCP of the two suffixes with rank $i$ and rank $j$. So, for the sufix array displayed above, $\mathrm{lcp}(4{,}5) = \mathrm{lcp}(5{,}4) = 2$. It is easy to prove by induction that the following property of suffix arrays holds: For any two ranks $i$ and $j$ with $j > i$, $\mathrm{lcp}(i,j)$ is equal to the smallest of the numbers $\mathrm{lcp}(i,i+1)$, $\mathrm{lcp}(i+1,i+2)$, …, $\mathrm{lcp}(j-1,j)$. We call this important property the **range minimum property** of LCP numbers. Note that without the premise of lexicographical ordering this smallest number would only be a lower bound of $\mathrm{lcp}(i,j)$.

The basic idea of how to find the shortest mismatch number of the character at position $a$ (henceforth, $\mathrm{smn}(a)$) in $T$ is to look at all suffixes with an index $b$ smaller than $a$ and to compare the values $\mathrm{lcp}(i,j)$ where $i$ is the rank of the suffix starting at $a$ and $j$ is the rank of the suffix starting at $b$. The largest of these values (which could be called the "longest prefix match") is incremented by one to obtain the shortest mismatch number. However, special care must be taken of the case of overlapping prefixes. A simple example is our sample text $T$ :=bananas which has, amongst others, the suffixes ananas (index 1, rank 0) and anas (index 3, rank 1). Even though the LCP of these two suffixes, viz. ana, has length $\mathrm{lcp}(0{,}1) = 3$, the shortest mismatch number at position 3 in the original string $T$ is not $3 + 1 = 4$, but only 3, as the above definition of the shortest mismatch number at position $a$ requires that the previous occurrences of substrings $T[a..b]$ to be considered for $\mathrm{smn}(a)$ must be contained completely within $T[0..(a-1)]$. In the example case, however, the two occurrences of the LCP ana overlap at the position of the second a, since the LCP length (3) is larger than the difference between the starting positions (indices) of the two suffixes, 2 in this case. In this **overlap situation**, the relevant number ist not the LCP length $\mathrm{lcp}(0{,}1)$, but the index difference $3 - 1 = 2$. Let us denote the index of the suffix with rank $r$ by $\mathrm{index}(r)$. In an overlap situation where $\mathrm{index}(i) > \mathrm{index}(j)$ and $\mathrm{lcp}(i,j) < \mathrm{index}(i) -$

$\mathrm{index}(j)$ the **effective LCP length** for the suffix ranks $i$ and $j$, henceforth $\mathrm{elcp}(i,j)$, is defined to be $\mathrm{index}(i) - \mathrm{index}(j)$. If $\mathrm{index}(j) > \mathrm{index}(i)$, we set $\mathrm{elcp}(i,j) \coloneqq 0$ since suffixes with a later starting position are irrelevant; in all other cases, $\mathrm{elcp}(i,j) \coloneqq \mathrm{lcp}(i,j)$.

In this terminology, our task of finding $\mathrm{smn}(a)$ finally boils down to finding the largest number $\mathrm{elcp}(i,j)$ such that $a = \mathrm{index}(i)$. Our algorithm splits this search into two subtasks.

- The first task sequentially examines the suffixes with higher ranks $j = i + 1$, $j = i + 2$, …, in the indicated ascending order and calculates, at each step, the value of $\mathrm{elcp}(i,j)$, using the range minimum property of LCP numbers for a stepwise calculation of $\mathrm{lcp}(i,j)$. As soon as a suffix $S$ with smaller index and no prefix overlap situation is found, the search can be cancelled, since no further suffixes may yield an even higher elcp-value than the ones encountered so far. This can be shown as follows: For any suffix with an index smaller than $a$ and with a rank $j$' even higher than the rank $j$ of $S$, $\mathrm{elcp}(i,j') \leq \mathrm{lcp}(i,j') \leq \mathrm{lcp}(i,j) = \mathrm{elcp}(i,j)$, where the last value must in turn be less than or equal to the largest effective LCP length found so far. This largest value of $\mathrm{elcp}(i,j)$ is returned as the result of the task.
- The second task sequentially examines the suffixes with lower ranks $k = i - 1$, $k = i - 2$, …, in the indicated descending order and proceeds in a fashion completely analogous to the first task to determine the largest number among the values of $\mathrm{elcp}(i,k)$.

At the end, the higher of the results obtained by the two tasks is incremented by one to get $\mathrm{smn}(a)$.

In order to obtain the smallest mismatch numbers for all character positions of the input text $T$, it is sufficient to iterate over all suffix ranks in arbitrary order and store, for each rank $i$, the results of the above procedure in an array that assigns to each index $a = \mathrm{index}(i)$ the value of $\mathrm{smn}(a)$.

*Notes on the general runtime characteristics of the tool*

The algorithm presented here solves the problem of calculating $\mathrm{smn}(a)$ for all positions (indexes) $a$ in a text $T$. A naïve algorithmic approach to this problem would consist in finding repetitions of substrings by brute-force substring search, which can be sped up considerably by caching the starting position of the first occurrence of substrings of $T$ in a dictionary-like structure. This approach fails on texts, even very short ones with only a few thousand characters, that have long repeated substrings, either because substring searches become too time-consuming or, in the case of caching, dictionary size explodes.

The concept of suffix arrays (Manber/Myers 1993) used instead belongs to the standard algorithmic toolkit of contemporary computer science (Sedgewick/Wayne 2011). As a major prerequisite for executing the algorithm, the suffix array and the corresponding LCP array have to be constructed, which can in principle be done in $O(n)$ (Kärkkäinen/Sanders 2003). Our implementation, however, uses a fast quicksort for the suffix array and constructs the LCP array afterwards by naïve substring comparison.

The algorithm for finding the largest $\mathrm{elcp}$-values in the suffix/LCP arrays just performs simple lookup and comparison operations, so no further string comparisons or searches within the suffix array are necessary. This implies that memory requirements both for storing and processing the suffix array are linear in the length of the text.

Although there is still room for improvement in the LCP array construction part (cf. Kasai et al. 2001), the tool is commendably fast for practical purposes, processing typical natural language texts of several megabytes length in a few seconds on average modern computers.

## References

Kärkkäinen, J.; Sanders, P. (2003). "Simple linear work suffix array construction". *Proceedings of the 30th international conference on Automata, languages and programming*, pp. 943-955.

Kasai, Toru; Lee, Gunho; Arimura, Hiroki; Arikawa, Setsuo; Park, Kunsoo (2001): Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, pp. 181-192.

Kontoyiannis, I.; Algoet, P.H.; Suhov, Y.M.; Wyner, A.J. (1998). "Nonparametric entropy estimation for stationary processes and random fields, with applications to English text." *IEEE Transactions on Information Theory* 44(3), pp. 1319-1327.

Manber, U.; Myers, G. (1993). "Suffix Arrays: A New Method for On-Line String Searches". *SIAM Journal on Computing* 22 (5), pp. 935-948.

Sedgewick, R.; Wayne, K. (2011). *Algorithms. Fourth edition*. Upper Saddle River etc.: Addison-Wesley.