

# WallBreaker - overcoming the wall effect in similarity search

Stefan Gerdjikov  
Faculty for Mathematics and  
Informatics  
Sofia University  
5 James Bourchier blvd.  
1164 Sofia, Bulgaria  
st\_gerdjikov@abv.bg

Stoyan Mihov  
Institute of Information and  
Communication Technologies  
Bulgarian Academy of Science  
Acad. G. Bonchev St., Block 25A  
1113 Sofia, Bulgaria  
stoyan@lml.bas.bg

Petar Mitankin  
Faculty for Mathematics and  
Informatics  
Sofia University  
5 James Bourchier blvd.  
1164 Sofia, Bulgaria  
pmitankin@fmi.uni-sofia.bg

Klaus U. Schulz  
Centrum für Informations- und  
Sprachverarbeitung  
Ludwig-Maximilians-  
Universität München  
Oettingenstr. 67  
80538 München, Germany  
schulz@cis.uni-muenchen.de

## ABSTRACT

In this paper we present the WallBreaker system for similarity search as used in the String Similarity Search/Join Competition, 2013, organized by the Humboldt University of Berlin [1].

We consider the problem of how to efficiently find for a given string  $P$  (pattern) all words  $W$  in a lexicon such that the distance between  $P$  and  $W$  does not exceed a given bound  $b$ . Classical solutions to this problem try to align  $P$  with suitable lexicon words in a strict left-to-right manner, starting at the left border of the pattern. During the search, only prefixes of lexicon words are visited where the distance to a prefix  $P'$  of the pattern does not exceed the given bound  $b$ . The main problem with this solution is the so-called “wall effect”: if we tolerate  $b$  errors and start searching in the lexicon from left to right, then in the first  $b$  steps we have to consider all prefixes of lexicon words. Eventually, only a tiny fraction of these prefixes will lead to a useful lexicon word, which means that our exhaustive initial search represents a waste of time.

To avoid the “wall effect”, in WallBreaker we have implemented our new method presented first in [3]. To sketch it let us assume that the pattern can be aligned with a lexicon word with not more than  $b$  errors. Clearly, if we divide the pattern into  $b + 1$  pieces, then at least one piece will exactly

match the corresponding substring of a lexicon word in the answer set. In our approach we first find the lexicon substrings that exactly match such a given piece of the pattern. Afterwards we continue by extending this alignment, step-wise attaching new pieces on the left or right side. For the alignment of new pieces, more errors are tolerated at each step, which guarantees that eventually  $b$  errors can occur. Since at later steps the set of interesting substrings to be extended is already small the wall effect is avoided, it does not hurt that we need to tolerate more errors. For this kind of search strategy, a new representation of the lexicon is needed where we can start traversal at any point of a word. In our new approach, the lexicon is represented as symmetric compact directed acyclic word graph (SCDAWG). This index structure can be seen as a part of a longer development of related index structures.

Our implementation executes the search queries in parallel. It is realized in ANSI C, compiled with GCC and does not use any additional libraries beside LIBC and POSIX threads. In average it performs a similarity search of a 100 character pattern with up to 16 errors in a lexicon with 750 000 entries in about 0.088 ms.

## 1. THE CONCEPT OF THE ALGORITHM

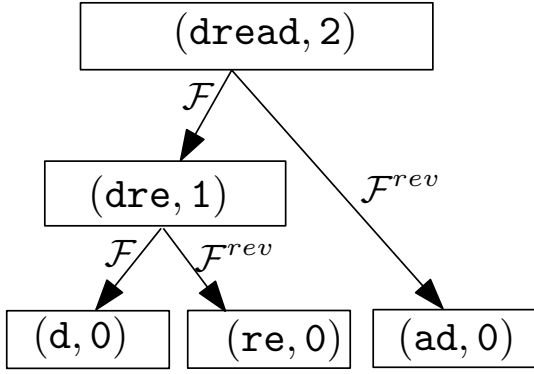
In this section we explain the idea of our algorithm using a small example. We also characterize the kind of resources needed to achieve its efficient implementation. Consider the dictionary

$$\mathcal{D} = \{\text{ear}, \text{real}, \text{lead}\}.$$

Suppose that for the pattern

$$P = \text{dread}$$

we want to find all words  $W$  in  $\mathcal{D}$  such that  $d_L(P, W) \leq 2$  where  $d_L$  is the Levenshtein edit-distance.



**Figure 1: Reducing the original query (dread, 2) into simpler ones. As a result we obtain an ordered binary tree representing search alternatives. The labels of the arcs indicate what sort of filter has to be used at the extension steps. The label  $\mathcal{F}$  shows that we extend to the right and thus an ordinary filter is required, whereas the label  $\mathcal{F}^{rev}$  means that we extend to the left and therefore a reverse filter has to supervise this step. The bound that determines a filter coincides with the threshold of the query written in the parent node.**

Let  $W$  in  $\mathcal{D}$  such that  $d_L(P, W) \leq 2$ . When we split  $P = \mathbf{dread}$  into the three parts  $\mathbf{d}$ ,  $\mathbf{re}$ ,  $\mathbf{ad}$ , then there must be a corresponding representation of  $W$  in the form  $W = W_1 \circ W_2 \circ W_3$  such that  $d_L(\mathbf{d}, W_1) + d_L(\mathbf{re}, W_2) + d_L(\mathbf{ad}, W_3) \leq 2$ . Then either  $d_L(\mathbf{d}, W_1) = 0$ , or  $d_L(\mathbf{re}, W_2) = 0$  or  $d_L(\mathbf{ad}, W_3) = 0$ .

In what follows, the notation  $(\mathbf{dread}, 2)$  is used as a shorthand for the algorithmic task to find all substrings  $V \in \text{Subs}(\mathcal{D})$  such that  $d_L(\mathbf{dread}, V) \leq 2$ , and similarly for other strings and bounds. The expression  $(\mathbf{dread}, 2)$  is called a *query* with query pattern  $\mathbf{dread}$  and bound 2. Now consider the query tree depicted in Figure 1. The idea is to solve the problems labeling the nodes in a bottom-up manner. To solve the problems  $(\mathbf{d}, 0)$ ,  $(\mathbf{re}, 0)$  and  $(\mathbf{ad}, 0)$  just means to check if  $\mathbf{d}$ ,  $\mathbf{re}$ , or  $\mathbf{ad}$  are substrings of lexicon words. We then solve problem  $(\mathbf{dre}, 1)$ . This involves two independent steps.

1. We look for extensions of the substring  $\mathbf{d}$  (as a solution of the left child in the tree) at the right.
2. We look for extensions of the substring  $\mathbf{re}$  (as a solution of the right child in the tree) at the left.

It is important to note that both extension steps are controlled using a Levenshtein filter, see Section 3, for bound 1 for  $P' = \mathbf{dre}$  (see Figure 1). As a result we obtain the single solution  $\mathbf{re}$  for the query  $(\mathbf{dre}, 1)$ . The next step in the bottom-up procedure looks at the root node  $(\mathbf{dread}, 2)$ . Solving this node again involves two independent steps.

1. We look for extensions of the substring  $\mathbf{re}$  (as a solution of the left child in the tree) at the right.
2. We look for extensions of the substring  $\mathbf{ad}$  (as a solution of the right child in the tree) at the left.

At this final step we cannot avoid the use of a Levenshtein filter for  $\mathbf{dread}$  and bound 2. We respectively obtain (1)  $\mathbf{rea}$ ,

real and (2)  $\mathbf{ead}$ ,  $\mathbf{lead}$ . Finally, we select those substrings among  $\{\mathbf{rea}, \mathbf{real}, \mathbf{ead}, \mathbf{lead}\}$  that are words in the dictionary  $\mathcal{D}$ . Thus we obtain that the solutions of the original query are **real** and **lead**.

The correctness of this approach and its formal outline can be found in [3].

**REMARK 1.1.** In order to efficiently realize a bottom-up subsearch of the form indicated above we need

1. an index structure that supports the following tasks:
  - (a) given a string  $V$ , efficiently decide if  $V$  represents a substring of a lexicon word,
  - (b) given a substring  $V$  of a lexicon word, give immediate access to all substrings of lexicon words of the form  $V \circ \sigma$  that add one letter  $\sigma \in \Sigma$  to the right,
  - (c) given a substring  $V$  of a lexicon word, give immediate access to all substrings of lexicon words of the form  $\sigma \circ V$  that add one letter  $\sigma \in \Sigma$  to the left.
2. A filter for the bound  $b$  specified at the parent node faced at an upward step. The filter takes as first input the query pattern  $P'$  specified at the parent node. Subsearches start with a given solution of the left (right) child query. When adding letters to the right (left) we use a conventional (“reversed”) filter.

We are going to consider these two issues in the following Sections.

## 2. SYMMETRIC COMPACT DIRECTED ACYCLIC WORD GRAPHS

In order to construct a data structure meeting the requirements 1a and 1b, we first need to represent the set of substrings of the lexicon,  $\mathcal{D}$ .

The number of different substrings in a lexicon  $\mathcal{D}$  can be proportional to the sum of the length squares of the words in  $\mathcal{D}$ . Nevertheless they can be represented via an automaton of size proportional to sum of lengths of the words in  $\mathcal{D}$ , [2, 4].

The key observation is that one needs to store (a representation) only of such substrings  $S \in \text{Subs}(\mathcal{D})$  which occur in different left and in different right contexts in  $\mathcal{D}$ , [4]. And these substrings are only a few, [2].

We used the algorithm of Inenaga et al. [4] in order to construct such an automaton in linear time and space. Essentially this automaton provides the solution required for the right extensions.

In [3], we also describe how this structure can be amended with additional information as to satisfy requirement 1a. Our algorithm runs in linear time and space.

As a result we obtain a data structure of linear size which satisfies the requirements 1b and 1c.

For our example from the previous Section 1, where  $\mathcal{D} = \{\mathbf{ear}, \mathbf{read}, \mathbf{lead}\}$  this structure is built for the set  $\#\mathcal{D}\$$  and is depicted on Figure 2.

## 3. FILTERS

In this section we turn our attention to requirement 1c. It asks for an efficient *filter* which controls that only words

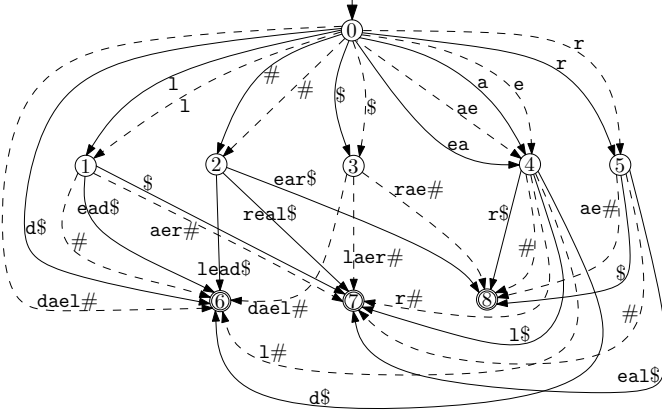


Figure 2: SCDAWG for  $\{\#ear\$, \#lead\$, \#real\}$ .

which have the chance to satisfy the query will be generated. In case of Levenshtein edit-distance this can be stated as follows [3]:

DEFINITION 3.1. Let  $b \in \mathbb{N}$  denote a given bound. By a Levenshtein filter for bound  $b$  we mean any algorithm that takes as input two words  $P, U \in \Sigma^*$  and decides

1. if there exists a string  $V \in \Sigma^*$  such that  $d_L(P, U \circ V) \leq b$ ,
2. if  $d_L(P, U) \leq b$ .

In what follows we briefly revisit three different kinds of Levenshtein filters which correspond to different trade-offs between space and time.

### 3.1 Ukkonen dynamic programming scheme

In [8] Ukkonen presents a dynamic programming scheme which given two words  $P$  and  $W$  and a bound  $b$  determines whether  $d_L(P, W) \leq b$  and if this is the case it also provides the edit-distance  $d_L(P, W)$ . The problem is reduced to the computation of a strip of length  $2b + 1$  along the main diagonal of a matrix of size  $|P| \times |W|$ . In this way the Ukkonen's algorithm requires only linear space but the time required by the algorithm is  $O(b \min(|P|, |W|))$ .

### 3.2 Levenshtein automata

The deterministic Levenshtein automata [7, 5, 6] are uniquely determined by a bound  $b$  and do not depend on the specific alphabet, see Figure 3. Their alphabet consists of bit vectors of length at most  $2b + 2$ . Given two words  $P$  and  $W$  the problem whether  $d_L(P, W) \leq b$  is reduced to the problem whether a sequence of bit vectors  $\chi(W, P)$  is recognised by the deterministic Levenshtein automaton determined by the bound  $b$ . The sequence of bit vectors  $\chi(W, P)$  is computed from  $W$  and  $P$  in linear time  $O(b|P|)$  and the traversal of the automaton then requires  $O(|W|)$  time. Furthermore if  $\chi(W, P)$  is recognised by the Levenshtein automaton, one can read the value  $d_L(P, W)$  from the final state reached by the traversal. Thus Levenshtein automata spare a constant factor in comparison to the Ukkonen's algorithm, but the space required for their storage grows exponentially with  $b$ .

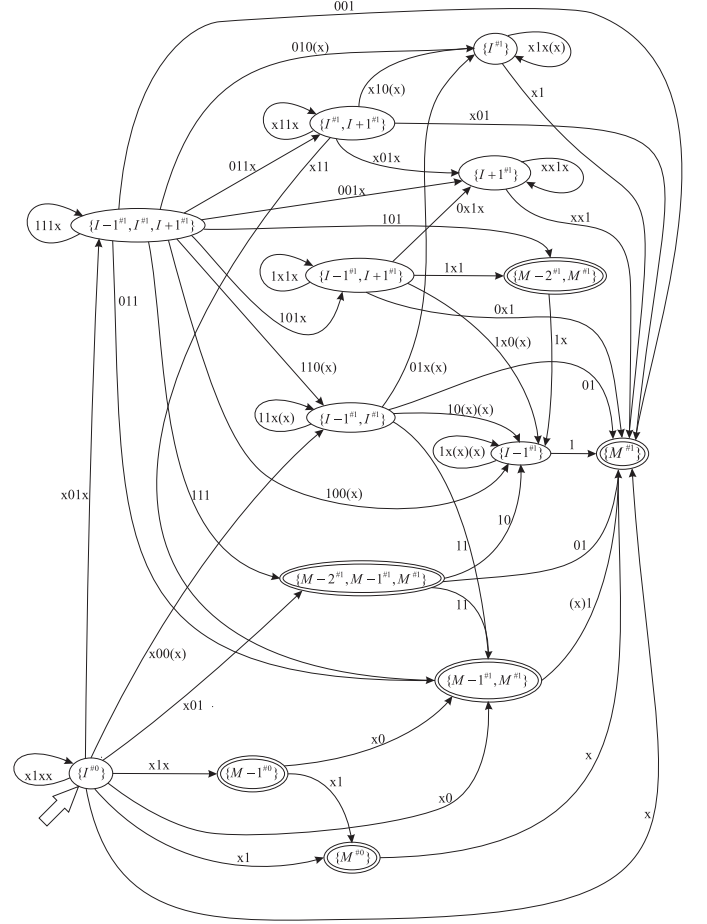


Figure 3: The deterministic Levenshtein automaton for  $b = 1$

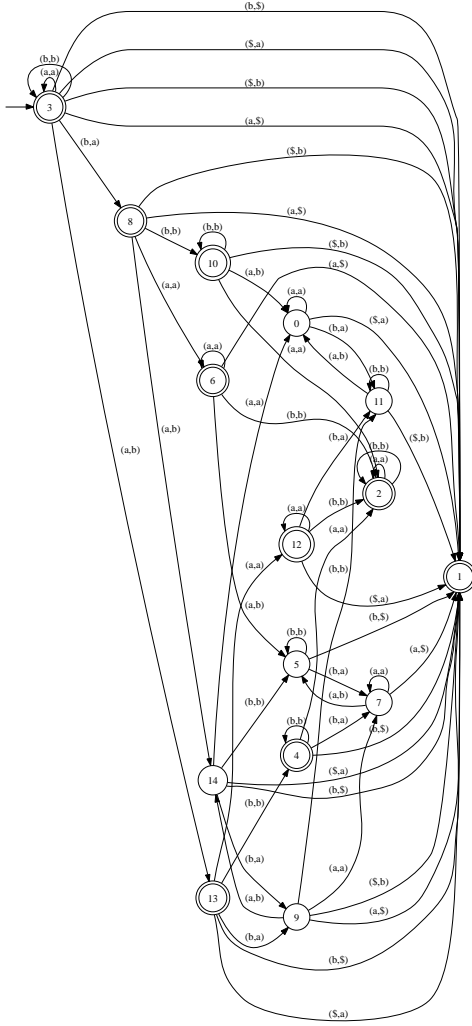


Figure 4: The synchronised Levenshtein automaton for  $b = 1$

### 3.3 Synchronised automata

Given a bound  $b$  and an alphabet  $\Sigma$ , the synchronised automata [6] recognise pairs of words  $\langle P, W \rangle$  over  $\Sigma$  such that  $d_L(P, W) \leq b$ , see Figure 4. The alphabet of the synchronised automaton is  $(\Sigma \cup \$) \times (\Sigma \cup \$)$  where  $\$$  is a new character and they are deterministic with respect to this alphabet. Thus the property  $d_L(P, W) \leq b$  can be tested in time  $O(\min(|P|, |W|) + b)$  and hence is a factor of  $b$  faster than the equivalent Levenshtein automaton or the Ukkonen’s algorithm. However being dependent on the size of the alphabet  $\Sigma$  they require much more space than the deterministic Levenshtein automaton for the same bound.

## 4. IMPLEMENTATION DETAILS

As explained in [3] the bottom-up search can be equivalently replaced by a depth-first search. This choice is determined by the space constraints during the search. In this way we need only  $O(|P|)$  additional space per query word  $P$ . However some of the answers of the query may be reported more than once.

WallBreaker is implemented in ANSI C and can be compiled with GCC. For the creation of the SCDAWG we do not use any additional libraries beside LIBC. For parallel processing of queries we use POSIX threads. The implementation was tested on Fedora 17 and Mac OS X 10.6.6.

## 5. EXPERIMENTAL RESULTS

We tested WallBreaker on the 5% excerpt of the genom reads provided by the organizers of the competition, [1], - 750 000 strings, average length per string - 100, alphabet size - 5. The experiments were run on a machine with 64 gigabytes of RAM, two 2.4 GHz Quad-Core Intel Xeon 4-core processors, 256 KB L2 cache memory per core and 12 MB L3 cache memory per processor. The construction of the SCDWAG took 324 seconds and the allocated memory was 2623.13 MB. We generated 100 000 query patterns by applying random insertions, deletions and substitutions to strings from the 5% excerpt of the genom reads. The number of errors was randomly chosen from the set  $\{0, 4, 8, 12, 16\}$ . The generated 100 000 patterns were processed for 8.794 seconds using 16 threads.

Further experimental results can be find in Section 6 in [3]. There the authors report tests conducted on different datasets and also communicate query times.

## 6. ACKNOWLEDGEMENTS

The research work reported in the paper is partly supported by the project CULTURA “CULTivating Understanding and Research through Adaptivity”, grant 269973, funded by the FP7 Programme (Specific Targeted Research Projects) and the project AComIn “Advanced Computing for Innovation”, grant 316087, funded by the FP7 Capacity Programme (Research Potential of Convergence Regions).

## 7. REFERENCES

- [1] <http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013>.
- [2] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the Association for Computing Machinery*, 34(3):578–595, 1987.
- [3] S. Gerdjikov, S. Mihov, P. Mitankin, and K. U. Schulz. Good parts first - a new algorithm for approximate search in lexica and string databases. *ArXiv e-prints*, Jan. 2013.
- [4] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. *Word Journal Of The International Linguistic Association*, 146(2):1–12, 2005.
- [5] S. Mihov and K. U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, 2004.
- [6] P. Mitankin, S. Mihov, and K. U. Schulz. Deciding word neighborhood with universal neighborhood automata. *Theoretical Computer Science*, 412(22):2340 – 2355, 2011.
- [7] K. U. Schulz and S. Mihov. Fast string correction with Levenshtein automata. *IJDAR*, 5(1):67–85, 2002.
- [8] E. Ukkonen. Algorithms for approximate string matching. *Information Control*, 64:100–18, 1985.