

# Scalable String Similarity Search/Join with Approximate Seeds and Multiple Backtracking

Enrico Siragusa<sup>\*</sup>  
Institute for Computer Science  
Freie Universität Berlin  
Takustr. 9, 14195  
Berlin, Germany  
enrico.siragusa@fu-berlin.de

David Weese  
Institute for Computer Science  
Freie Universität Berlin  
Takustr. 9, 14195  
Berlin, Germany  
david.weese@fu-berlin.de

Knut Reinert  
Institute for Computer Science  
Freie Universität Berlin  
Takustr. 9, 14195  
Berlin, Germany  
knut.reinert@fu-berlin.de

## ABSTRACT

We present in this paper scalable algorithms for optimal string similarity search and join. Our methods are variations of those applied in *Masai* [15], our recently published tool for mapping high-throughput DNA sequencing data with unprecedented speed and accuracy. The key features of our approach are filtration with approximate seeds and methods for multiple backtracking. Approximate seeds, compared to exact seeds, increase filtration specificity while preserving sensitivity. Multiple backtracking amortizes the cost of searching a large set of seeds. Combined together, these two methods significantly speed up string similarity search and join operations. Our tool is implemented in C++ and OpenMP using the SeqAn library. The source code is distributed under the BSD license and can be freely downloaded from <http://www.seqan.de/projects/edbt2013>.

## Categories and Subject Descriptors

H.3.0 [Information Storage and Retrieval]: General

## General Terms

Algorithms

## Keywords

banded dynamic programming, banded Myers bit-vector, radix tree, suffix tree, backtracking, filtration, approximate seeds

## 1. INTRODUCTION

This work was inspired by the *String Similarity Search/Join competition* initiated by the group of Ulf Leser at the Humboldt University in Berlin, Germany.

<sup>\*</sup>To whom correspondence should be addressed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT/ICDT '13, March 18 – 22 2013, Genoa, Italy

Copyright 2013 ACM 978-1-4503-1599-9/13/03 ...\$15.00.

## 1.1 Competition

The competition consists of performing string similarity operations under *unweighted edit distance* for different error thresholds  $k$ . The competition considers two tracks:

- I String Similarity Search (SEARCH): Given a set of strings  $S$  and a set of query strings  $Q$ , for each query  $Q \in Q$  find all strings in  $S$  within distance  $k$ .
- II String Similarity Join (JOIN): Given a set of strings  $S$ , find all pairs of strings within distance  $k$ .

Clearly, JOIN of a set  $S$  is equivalent to SEARCH  $Q$  in  $S/Q$ , for all  $Q \in S$ . Consequently, in the following of this paper, unless explicitly stated, we consider only SEARCH.

## 1.2 Our contribution

We present in this paper efficient and practical algorithms for solving SEARCH and JOIN. Our methods are variations of those applied in *Masai* [15], our recently published tool for mapping high-throughput DNA sequencing data with unprecedented speed and accuracy.

Masai is based on efficient filtering methods for approximate string matching, namely *approximate seeds* and *multiple backtracking*. Approximate seeds provide full-sensitive filtration without sacrificing filtration specificity. Multiple backtracking speeds up filtration by searching all seeds simultaneously with the help of an additional index. In [15] we showed that, combined together, these methods yield a flexible and efficient filter that significantly speeds up approximate search on genomic data sets.

We show in this paper that minor variations of the methods of [15] are sufficient to solve efficiently SEARCH and JOIN. Differently from [15], we propose:

- multiple backtracking under the *edit distance*;
- a parallelization of multiple backtracking;
- stricter filtration criteria for the  $k$ -difference global alignment problem.

In Section 2, we consider online methods for edit distance computation, first presented in [18]. Then, in Section 3, we consider backtracking methods on indices, generalizing those presented in [15]. Finally, in Section 4, we propose variants of the filtering methods applied in [15], and based on the previous two methods.

## 2. ONLINE SEARCH

In the following, we first define the edit distance between two strings and then describe different online algorithms to compute it. Finally, we propose a banded variant of Myers' bit-parallel algorithm.

### 2.1 Banded DP

Edit distance computation is a well-known problem and studied in many publications. For two given strings, the edit distance is the minimal number of required edit operations to transform one string into the other, where an operation can be a deletion, an insertion, a replacement, or a match (no change). Sellers [14] proposed an alignment algorithm to compute the edit distance between two strings of length  $m$  and  $n$  in  $\mathcal{O}(mn)$  time, w.l.o.g. we assume  $m < n$ . The algorithmic idea goes back to the more general approach by Needleman and Wunsch [13] which computes alignments with maximal similarity in  $\mathcal{O}(mn^2)$  time allowing for arbitrary gap costs that depend on the gap length. Both algorithms are based on dynamic programming (DP) and compute the cells of an  $m \times n$  DP matrix.

More specialized is the problem to compute the edit distance only up to a maximal distance  $k$  and to determine that it is above  $k$ , otherwise. This problem is called the *banded* or *k-difference global alignment problem*. We refer the reader to Gusfield [7] for more details.

**LEMMA 1.** *The  $k$ -difference global alignment problem can be solved by computing only a diagonal band of the DP matrix of width  $k+1$ , where the leftmost band diagonal is  $\lfloor \frac{m-n+k}{2} \rfloor$  cells left of the main diagonal (see Figure 1).*

**PROOF.** Indirect. Assume that a cell outside the band is part of a global alignment with at most  $k$  errors. If the cell is left of the band, the traceback that starts in the top left corner would go down at least  $c = \lfloor \frac{m-n+k}{2} \rfloor + 1$  cells. Then it needs to go right at least  $n - m + c$  cells to end in the bottom right corner. Hence it contains at least  $n - m + 2c > n - m + 2 \frac{m-n+k}{2} = k$  errors. The assumption that the cell is right of the band can be falsified analogously.  $\square$

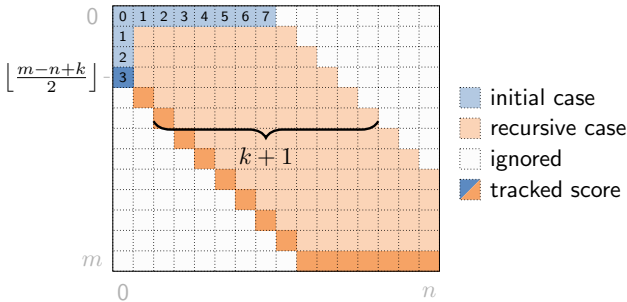


Figure 1: Banded  $k$ -difference global alignment between two strings of length  $m$  and  $n$ . The traceback of every possible global alignment with up to  $k$  errors resides completely in the colored area. Hence the white cells need not to be computed.

### 2.2 Banded Myers bit-vector algorithm

In [11], Myers introduced an algorithm that exploits bit-parallelism to compute the edit distance in  $\mathcal{O}(\frac{mn}{w} + m \cdot |\Sigma|)$  time, where  $w$  is the CPU word size and  $\Sigma$  the string alphabet. The fundamental idea of his approach is to represent a

DP column by the score in the last row and 2 bit-vectors of size  $m$  encoding the vertical differences -1, 0, or 1 of adjacent cells. The DP matrix is computed column-wise, where the next column can then be computed by  $\mathcal{O}(\frac{m}{w})$  logical and arithmetical operations on these bit-vectors.

In [18; 15] we proposed a modification of Myers' algorithm that computes only the columns of a diagonal band of the DP matrix yielding the practically fastest algorithm for the  $k$ -difference global alignment problem. Whereas Myers' original algorithm tracks the scores along the last DP row and computes bit-vectors spanning whole DP columns, our approach tracks the scores of the lowest band cells (dark cells in Figure 1) and computes bit-vectors spanning only the intersection of band and DP columns. We dispensed with the obligation to preprocess the pattern and reduced the running time from  $\mathcal{O}(\frac{m(n+|\Sigma|)}{w})$  to  $\mathcal{O}(\frac{(k+1)(n+|\Sigma|)}{w})$  and the overall memory consumption from  $\mathcal{O}(m|\Sigma|)$  to  $\mathcal{O}((k+1)|\Sigma|)$  bits.

## 3. BACKTRACKING

A naïve approach for solving SEARCH is to compute all  $|Q| \cdot |\mathcal{S}|$   $k$ -difference global alignments using online search. No matter how fast online search can be, this naïve approach quickly becomes impractical. Since the database  $\mathcal{S}$  is static and given in advance, we decide to index it. To this intent, we first introduce *radix trees*, optimal data structures representing sets of strings. Later on, we consider algorithms solving *multiple k-difference global alignment problem* instances on radix trees.

### 3.1 Radix trees

The radix tree [10] is a lexicographically ordered tree data structure representing a set of strings. It can be built via radix sort in time and space linear in the total length of the strings.

Assume w.l.o.g. that each string in  $\mathcal{S}$  is padded with a distinct *terminator symbol*, not being part of the string alphabet  $\Sigma^1$ . The radix tree of  $\mathcal{S}$  has one node designated as the root, and one leaf per string in the set. Every internal node has more than one child, and edges are labeled with non-empty strings. Each path from the root to an internal node spells a different unique prefix. Consequently, prefixes common to distinct strings in  $\mathcal{S}$  are compressed.

For simplicity of exposition, in the following we consider *tries*<sup>2</sup>, although our algorithms are easily extendable to work on trees. Hence, in the following, we assume  $\mathcal{S}$  and  $\mathcal{Q}$  to be indexed using radix tries  $\mathbb{S}$  and  $\mathbb{Q}$ . Given a node  $x$ , we denote with  $label(x)$  the label of the edge entering into  $x$ , with  $\mathcal{C}(x)$  the set of children of  $x$  being internal nodes<sup>3</sup>, with  $\mathcal{E}(x)$  the set of children of  $x$  being leaves<sup>4</sup>.

Using a radix tree we can find all strings in  $\mathcal{S}$  equal to a query string  $Q$ , in optimal time  $\mathcal{O}(|Q|)$  and independently of  $|\mathcal{S}|$ . We locate a query  $Q$  by starting in the root node of  $\mathbb{S}$  and following the path spelling the query. If we end up in a node  $x$ , each leaf  $l_x \in \mathcal{E}(x)$  points to a distinct string  $S_x \in \mathcal{S}$  that is equal to  $Q$ .

<sup>1</sup>Terminator symbols are necessary to ensure that no string  $S_i \in \mathcal{S}$  is a prefix of another string  $S_j \in \mathcal{S}$ .

<sup>2</sup>On tries, internal nodes can have only one child.

<sup>3</sup>Entering edges of internal nodes are labeled with symbols in  $\Sigma$ .

<sup>4</sup>Entering edges of leaves are labeled with terminator symbols.

By backtracking [16; 2] on a radix tree we can find all strings in  $\mathcal{S}$  within distance  $k$  from a query string  $Q$ , in average time sublinear in  $|\mathcal{S}|$  [12]. A top-down traversal on the radix tree  $\mathbb{S}$  spells incrementally all distinct strings present in the set  $\mathcal{S}$ . While traversing each branch of the tree, we incrementally compute the distance between the query and the spelled string. If the computed distance exceeds  $k$ , we stop the traversal and proceed on the next branch. Conversely, if we completely spelled the query  $Q$ , and we ended up in a node  $x$ , each leaf  $l_x \in \mathcal{E}(x)$  points to a distinct string  $S_x \in \mathcal{S}$  that is within distance  $k$  of  $Q$ .

### 3.2 Multiple backtracking

In SEARCH we are given a set of query strings  $\mathcal{Q}$ , i.e. we are given several query strings at the same time. Consequently, we decide to index also  $\mathcal{Q}$ . Note that, in JOIN, the radix tree of queries  $\mathcal{Q}$  is equal to the radix tree of  $\mathcal{S}$ .

To this intent, we now introduce methods for *multiple offline approximate search* to simultaneously locate a set of queries  $\mathcal{Q}$  in a set of strings  $\mathcal{S}$ . We start with an algorithm for multiple offline exact search, and later we extend it to multiple approximate search.

#### 3.2.1 Exact search

Algorithm 1 takes as input two nodes  $s$  and  $q$ , respectively, of  $\mathbb{S}$  and  $\mathbb{Q}$ , and reports all pairs of leaves  $(l_s, l_q) \in \mathcal{L}(s) \times \mathcal{L}(q)$  such that the path from  $q$  to  $l_q$  spells exactly the path from  $s$  to  $l_s$ . Consequently, by applying Algorithm 1 on the root nodes of  $\mathbb{S}$  and  $\mathbb{Q}$ , we obtain all pairs of leaves  $(l_s, l_q)$  such that the query string pointed by  $l_q$  is equal to the string pointed by  $l_s$ .

---

**Algorithm 1** Multiple exact search.

---

```

1: procedure SEARCH( $s, q$ )
2:   report  $\mathcal{E}(s) \times \mathcal{E}(q)$ 
3:   for all  $c_q \in \mathcal{C}(q)$  do
4:     if  $\exists c_s \in \mathcal{C}(s) : \text{label}(c_s) = \text{label}(c_q)$  then
5:       SEARCH( $c_s, c_q$ )
6:     end if
7: end procedure

```

---

In JOIN, since  $\mathbb{Q} = \mathbb{S}$ , Algorithm 1 can be replaced by a simple top-down traversal that reports each leaf paired to each of its sibling leaves.

#### 3.2.2 Approximate search

Algorithm 2 takes an additional input argument  $e$  denoting the number of errors left, i.e. initially  $e = k$ . The algorithm computes the union of all paths in the subtrees rooted in  $s$  and  $q$ , within edit distance  $k$ . It reports all pairs of leaves  $(l_s, l_q) \in \mathcal{L}(s) \times \mathcal{L}(q)$  such that the path from  $q$  to  $l_q$  spells the path from  $s$  to  $l_s$  within edit distance  $k$ .

Therefore, by applying Algorithm 2 on the root nodes of  $\mathbb{S}$  and  $\mathbb{Q}$ , we obtain all pairs of leaves  $(l_s, l_q)$  such that the query string pointed by  $l_q$  is within edit distance  $k$  from the string pointed by  $l_s$ .

For  $k = 0$ , lines 5–16 of Algorithm 2 are equivalent to Algorithm 1. However Algorithm 1 is preferred to Algorithm 2 because it traverses only edges spelling common strings instead of all pairs of edges and it is thus more efficient. Figure 2 in [15] depicts a run of Algorithm 2.

---

**Algorithm 2** Multiple approximate search.

---

```

1: procedure SEARCH( $s, q, e$ )
2:   if  $k = 0$  then
3:     SEARCH( $s, q$ )
4:   else
5:     report  $\mathcal{E}(s) \times \mathcal{E}(q)$ 
6:     for all  $c_s \in \mathcal{C}(s)$  do
7:       for all  $c_q \in \mathcal{C}(q)$  do
8:         if  $\text{label}(c_s) = \text{label}(c_q)$  then
9:           SEARCH( $c_s, c_q, e$ )
10:        else
11:          SEARCH( $c_s, c_q, e - 1$ )
12:        end if
13:      for all  $c_s \in \mathcal{C}(s)$  do
14:        SEARCH( $c_s, q, e - 1$ )
15:      for all  $c_q \in \mathcal{C}(q)$  do
16:        SEARCH( $s, c_q, e - 1$ )
17:    end if
18: end procedure

```

---

#### 3.2.3 Parallelization

Parallelization of multiple backtracking is not straightforward. We start multiple backtracking with a single thread on the top of the two trees. Every time the traversal reaches a fixed depth  $d$ , instead of recursing, we append the arguments of recursive calls to Algorithms 1 and 2 to a work queue. In this way, we save the state of the algorithms at depth  $d$ . Each saved recursive call defines an independent job.

Once the algorithm stops, we continue backtracking in parallel. We let a fixed number of threads remove recursive call arguments from the queue, and continue the recursion by calling Algorithms 1 or 2 with removed call arguments.

To facilitate load balancing, the work queue must contain an adequate number of jobs. We choose  $d$  such that the work queue contains roughly 1000 jobs.

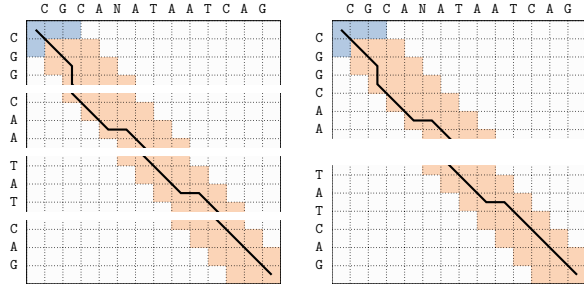
## 4. FILTRATION

Backtracking methods presented in Section 3 are only practical for short queries and small edit distance. For instance, single backtracking for a query  $Q$  exhibit worst case-time complexity  $\mathcal{O}(|Q|^k \cdot |\Sigma|^k)$  [12]. We can obtain more efficient and practical filtration methods by combining the methods presented in Section 2 and 3.

The banded DP alignment matrix in Figure 2 suggests us the key idea behind filtration methods. Let  $Q$  be one query string, and  $S$  any string from the set  $\mathcal{S}$ . We partition the query string  $Q$  in non-overlapping factors called seeds. Such partitioning induces a partitioning of the DP alignment matrix of  $Q$  and  $S$ , and a partitioning of the global alignment trace in smaller traces. We are only interested into occurrences of seeds from  $Q$  in  $S$  that would fall inside the band of the DP alignment matrix. Indeed, seeds occurring outside of the band would not be part of any valid global alignment trace.

### 4.1 Exact seeds

An application of the pigeonhole lemma, analogous to that one proposed in [3], gives us a convenient partitioning. We partition a query string  $Q$  into  $k + 1$  non-overlapping seeds. Since each edit operation can affect at most one seed, for



(a) Exact seeds. (b) Approximate seeds.

Figure 2: **Filtration strategies.** A query string  $Q$  (vertical) matches a database string  $S$  (horizontal) with 3 errors. (a) If we partition  $Q$  into 4 seeds, at least one seed (CAG) occurs exactly in  $S$ . (b) Alternatively, if we partition  $Q$  into 2 seeds, at least one seed (TATCAG) occurs with at most 1 error in  $S$ .

the pigeonhole principle, the trace of  $Q$  and  $S$  contains the trace of some seed without errors. Therefore, it is sufficient to search exactly all  $k + 1$  seeds of  $Q$  in the index of  $S$ , and to consider only their occurrences inside the diagonal band defined in Lemma 1. However, the converse is not true, therefore it is necessary to verify whether the occurrence of some seed of  $Q$  induces a string  $S$  within edit distance  $k$ .

Filtration specificity in terms of strings to verify is strongly correlated to seed length. Since we want to maximize the length of the shortest seed, we let the minimum seed length be  $\lfloor |Q|/(k+1) \rfloor$ . Clearly, a big  $k$  or a short query  $Q$  deteriorates filtration specificity. If we want to improve on it by increasing the seed length, we have to resort to approximate seeds.

## 4.2 Approximate seeds

A generalization of the pigeonhole lemma, analogous to that one proposed in [12], suggests us a more flexible partitioning. We partition  $Q$  into  $s \leq k+1$  non-overlapping seeds. According to the pigeonhole principle, the trace of  $Q$  and  $S$  then contains the trace of some seed within distance  $\lfloor k/s \rfloor$ . It is sufficient to search  $(k \bmod s) + 1$  seeds within distance  $\lfloor k/s \rfloor$ , and the remaining seeds within distance  $\lfloor k/s \rfloor - 1$ . To prove full-sensitivity it suffices to see that, if none of the seeds occur within its assigned distance, the total distance must be at least  $s \cdot \lfloor k/s \rfloor + (k \bmod s) + 1 = k + 1$ . Hence all strings  $S \in \mathcal{S}$  within distance  $k$  will be found.

Strings in  $Q$  have variable length and, in SEARCH, even variable number of errors. For simplicity, we decide to fix a priori the seed length  $l$ , and let it be a parameter to the filtration algorithm. The seed length  $l$  enforces the number of seeds  $s$  to be  $\lfloor |Q|/l \rfloor$  for each query string. The optimal seed length  $l$  depends on the nature of the database as well as on the query length and the absolute number of errors. In general, by increasing  $l$ , filtration becomes more specific at the expense of a higher filtration time.

## 4.3 Finding seeds

For simplicity, we index  $S$  with a *generalized suffix tree*  $\mathcal{S}$ . The suffix tree [19] of a string  $S$  is the radix tree of all the suffixes of  $S$ . It can be built in time and space  $\mathcal{O}(|S|)$  [17]. In addition, the *generalized suffix tree* is the suffix tree of a set of strings  $S$ . Once more, we refer the reader to Gusfield

[7] for more details.

Consequently, we partition each query string into a variable number of seeds of fixed length  $l$ , having a variable number of errors  $e \in [0, k]$ . We group all seeds (from all queries) having  $e$  errors in a set  $Q_e$ . Then, we index each group of seeds  $Q_e$  using one radix tree  $\mathcal{Q}_e$ .

In order to locate all seeds in  $Q_e$  as *substrings* of any string in  $\mathcal{S}$ , we apply Algorithm 1 and 2 to the root nodes of  $\mathcal{S}$  and  $\mathcal{Q}_e$ , and let them report  $\mathcal{L}(s) \times \mathcal{E}(q)$ , where  $\mathcal{L}(s)$  is the set of leaves of the subtree rooted in  $s$ .

## 5. IMPLEMENTATION

For simplicity, we emulate radix and suffix trees using *radix* and *suffix arrays* [9]. We implemented a generic suffix tree top-down traversal on top of the suffix array. Consequently, the implementation of our algorithms is abstract from that of the underlying substring index. Therefore, suffix arrays could be easily replaced by *enhanced suffix arrays* [1], *lazy suffix trees* [6], or *q-gram indices*.

Generalized suffix arrays can be constructed in linear time using an adaptation of the DC7 algorithm [4] to multiple sequences. Likewise, radix arrays can be constructed in linear time by radix sort. However, on one hand, we do not dispose of parallel versions of DC7 and radix sort. On the other hand, strings from SEARCH and JOIN test datasets have short longest common prefixes on average. For these reasons, we prefer to construct radix and suffix arrays using *std::sort* from the *Multi-Core Standard Template Library* (MCSTL).

Our implementation of multiple backtracking is more involved than the pseudocode of Algorithm 2. Indeed, we align tree labels using banded DP. Conversely, Algorithm 2 follows independently all active DP diagonals in the trees, analogously to [8]. Essentially, our implementation uses DP to simulate a *Nondeterministic Finite Automaton* (NFA) for the  $k$ -differences problem, while Algorithm 2 simulates the same NFA naively. Consequently, we prefer the former approach, since it minimizes the number of node traversals. A third possible approach simulates the NFA using bit-parallelism, as done in [12]. However, in [12] the full query string is preprocessed in advance, while in multiple backtracking this is not possible.

Our tool is implemented in C++ and OpenMP using the SeqAn [5] library. The source code is distributed under the BSD license and can be freely downloaded from <http://www.seqan.de/projects/edbt2013>. Online search, index construction, and multiple backtracking algorithms are part of the SeqAn library.

## References

- [1] Abouelhoda, M., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, **2**(1), 53–86.
- [2] Baeza-Yates, R. A. and Gonnet, G. H. (1999). A fast algorithm on average for all-against-all sequence matching. In *SPIRE/CRIWG*, pages 16–23. IEEE.
- [3] Baeza-Yates, R. A. and Navarro, G. (1999). Faster approximate string matching. *Algorithmica*, **23**(2), 127–158.
- [4] Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. (2008). Better external memory suffix array construction. *J. Exp. Algorithmics*, **12**, 3.4:1–3.4:24.

- [5] Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, **9**, 11.
- [6] Giegerich, R., Kurtz, S., and Stoye, J. (2003). Efficient implementation of lazy suffix trees. *Software Pract. Exper.*, **33**(11), 1035–1049.
- [7] Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: Computer science and computational biology*. Cambridge University Press, New York, NY, USA.
- [8] Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, **25**(14), 1754–1760.
- [9] Manber, U. and Myers, G. (1990). Suffix arrays: a new method for on-line string searches. In *SODA*, pages 319–327.
- [10] Morrison, D. R. (1968). PATRICIA-Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, **15**(4), 514–534.
- [11] Myers, G. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, **46**(3), 395–415.
- [12] Navarro, G. and Baeza-Yates, R. (2000). A hybrid indexing method for approximate string matching. *J. Discrete Algorithms*, **1**(1), 205–239.
- [13] Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, **48**, 443–453.
- [14] Sellers, P. H. (1980). The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, **1**(4), 359–373.
- [15] Siragusa, E., Weese, D., and Reinert, K. (2013). Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Res.* <http://nar.oxfordjournals.org/content/early/2013/01/28/nar.gkt005>.
- [16] Ukkonen, E. (1993). Approximate string-matching over suffix trees. In *CPM*, pages 228–242.
- [17] Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, **14**(3), 249–260.
- [18] Weese, D., Holtgrewe, M., and Reinert, K. (2012). RazerS3: faster, fully sensitive read mapping. *Bioinformatics*, **28**(20), 2592–2599.
- [19] Weiner, P. (1973). Linear pattern matching algorithms. In *SWAT (FOCS)*, pages 1–11. IEEE.