Using Tries for Evaluating Monge-Elkan Distance on Genomic Sequences

Petr RYŠAVÝ[1]

[1]Dept. of Computer Science, Czech Technical University, Technická 2, 166 27 Praha, Czech Republic

petr.rysavy@fel.cvut.cz

Abstract. *Processing of genetic sequences become a popular task in past years. Next-generation sequencing machines produce sets of short substrings of a DNA sequence, called reads. Recently there has been proposed a method how to measure distance between read sets using Monge-Elkan similarity known from the field of databases. In this paper we study applicability of tries on Monge-Elkan similarity. Tries have been used with success for improving runtime of dictionary search and similarity joins, which are problems similar to evaluating Monge-Elkan similarity. We show that our approach outperforms the straightforward evaluation of Monge-Elkan similarity for very short reads. However the speedup achieved is smaller than on dictionary search and similarity joins.*

Keywords

Monge-Elkan similarity, edit distance, trie, read bag

1. Introduction

The main source of evolutionary information in living beings is their DNA. It encodes everything that any animal, human or plant needs for proper function of proteins. Therefore one of the most popular tasks among scientists is to read this code and try to understand it. With a rapid development of sequencing technologies there are however higher demands on computing algorithms that are able to process genomic data.

Modern sequencing machines are not able to read a long sequence, but instead they read only short substrings of this sequence, called *reads*. Their length is usually from tens to hundreds of symbols. In our case the alphabet that forms genomic sequences contains only four letters $\Sigma = \{\mathsf{A}, \mathsf{T}, \mathsf{C}, \mathsf{G}\}$. Those 4 symbols correspond to 4 nucleotides that encode genetic information in DNA.

Because reads are short most of the work needs to be done in silico. Namely the task is left on assembly algorithms (for example [11, 4, 15]). Their goal is to reconstruct the original sequence from bag of reads. When an estimate of the sequence is built, one may try to calculate similarity between the sequences. One possible application is reconstruction of phylogenetic tree. Common similarity measure is Levenshtein distance [5], that measures minimum number of edits (insertions, deletions and substitutions) that are needed to transform one sequence into the other.

However the problem of sequence assembly is known to be NP-hard and therefore several approximation approaches have been proposed. As an alternative, one may try to avoid assembly step at all. This idea was first stated in [14] for purposes of supervised learning. Paper [9] proposed a way how to estimate similarity between two sequences knowing only their bags of reads. This approach is based on evaluating the Monge-Elkan distance [6], which is known from the field of databases.

If A is one of the original sequences, then R_A denotes a *read bag*, which is a multiset of $|R_A|$ reads sampled i.i.d. with replacement from the original sequence A. Moreover we assume that length of all reads is a fixed constant l, which comes from the sequencing machine setting. Let $\mathrm{dist}(A, B)$ be the Levenshtein distance between sequences A and B. We define Monge-Elkan distance as

$$\underset{\mathsf{ME}}{\mathrm{Dist}}(R_A, R_B) = \frac{1}{|R_A|} \sum_{a_i \in R_A} \min_{b_j \in R_B} \mathrm{dist}(a_i, b_j). \quad (1)$$

In paper [9] a symmetric version of the distance is used

$$\underset{\mathsf{MES}}{\mathrm{Dist}}(R_A, R_B) = \frac{1}{2}\left(\underset{\mathsf{ME}}{\mathrm{Dist}}(R_A, R_B) + \underset{\mathsf{ME}}{\mathrm{Dist}}(R_B, R_A)\right). \quad (2)$$

In Sect. 3 we propose a method how to use trie [1] for improving runtime of Monge-Elkan distance evaluation on read bags. Namely our goal is to provide speedup for method provided in [9]. Tries were used with success for problems of dictionary search and similarity joins, which are problems similar to Monge-Elkan similarity. We will overview approaches for solving those two related problems in Sect. 2. In Sect. 4 we will see that for very short reads our approach outperforms the straightforward implementation of Monge-Elkan distance in terms of runtime. However in Sect. 5 we will have to conclude that the speedups are not high enough to be the only approach that is used for evaluation of (2).

2. Related Work

Trie [1] is a data structure that can be used for storing sets of strings by exploiting common prefixes. It is efficient especially on small alphabets. Each node has up to $|\Sigma|$ children, each representing one single character prefix. Each leaf represents a string consisting of characters that can be found on the path from the root to the leaf. Because common prefixes are represented by the same path, trie forms a compact representation of a set of strings.

Paper [10] proposed a way how to use PATRICIA trie [7], an even more compact representation of trie, for *dictionary search*[1]. Given a string s, a set S and a threshold k, goal is to find $\{s' \in S \mid \mathrm{dist}(s, s') \leq k\}$. Suppose that query string is ATCA and $S = \{\mathsf{AGCT}, \mathsf{AGAA}\}$. For Levenshtein distance the standard Wagner-Fischer [13] dynamic programming algorithm is used. Once distance of ATCA and AGCT is calculated, the algorithm moves to calculating distance of ATCA and AGAA. However strings AGCT and AGAA share a common prefix — AG. The first three rows of the dynamic programming table will be equal. Therefore [10] proposes to build trie on set S and then traverse the trie in a way that avoids repeated work. Each time a leaf node of trie is reached, algorithm backtracks only to the deepest node lying on a path to an not-yet-explored string from set S.

Paper [8] enhances approach from [10] by filtering techniques and it also proposes to use trie for *similarity join*. Given two sets S_1 and S_2, goal is to find $\{(s_1, s_2) \in S_1 \times S_2 \mid \mathrm{dist}(s_1, s_2) \leq k\}$. The authors propose to build tries for both sets and then to traverse both tries concurrently to find similar tuples. Other approaches that use trie for similarity joins include [2, 3].

Similarity joins and dictionary searches are similar to evaluation of equation (2). However there are several differences that make the problem of evaluating (2) more difficult. First for each string in both sets we need to find its closest match regardless whether the final distance is small or large. Therefore in our approach we need to get to each leaf of trie at least once. Second, our motivation is to improve runtime of methods proposed in [9]. However in [9], the Levenshtein distance in (2) is replaced by a modified version, that makes traditional approaches for runtime improvements not applicable. This includes Ukkonen's cutoff [12] that allows algorithm to fill only few entries around the table diagonal.

3. Proposed Method

In this section we propose to use two tries to iterate over the read bags in order to evaluate formula (2). However instead of approach used in [8] we use a different trie representation. Each trie is represented by a triple of arrays. Each entry in those arrays corresponds to a node of a trie. Index of a node is determined based on assumption that any node has lower index then all its children. Either *preorder* traversal can be used or as in our case *BFS* traversal is appropriate.

In `parent` array we store pointer to the parent node. `Character` array contains information about which character labels traversal from parent to the node. Third array named `count` is used only for leaf nodes and stores information about count of read in the bag. Recall that all reads are supposed to have the same length l. Therefore the `count` array contains useful information only for the last layer of trie.

With this representation we can formulate the basic algorithm. We use the same approach as standard Wagner-Fischer algorithm [13] for calculating Levenshtein distance [5]. The only difference is that the algorithm looks on parent node in trie and instead of comparing characters in strings, it compares labels in trie.

When the whole table is filled, algorithm needs to look on the entries that correspond to Cartesian product of terminal nodes of the tries. For each row and column we need to find a miniminum corresponding to $\min_{b_j \in R_B}$ in (1). Then we multiply by the count of the corresponding read and finally we average over both directions.

Algorithm 1 Pseudocode for the proposed method

function SYMMONGE-ELKANDISTANCE(R_A, R_B)
  $T_A \leftarrow$ GETTRIE(R_A), $T_B \leftarrow$ GETTRIE(R_B)
  $arr \leftarrow$ empty 2D array of size $|T_A| \times |T_B|$
  $arr[0][0] \leftarrow 0$
  for $i \in \{1, 2, \ldots, |T_A|\}$ **do** ▷ initialize first column
    $arr[i][0] \leftarrow arr[T_A.par(i)][0] + 1$
  end for
  for $j \in \{1, 2, \ldots, |T_B|\}$ **do** ▷ initialize first row
    $arr[0][j] \leftarrow arr[0][T_B.par(j)] + 1$
  end for
  for $i \in \{1, 2, \ldots, |T_A|\}$ **do** ▷ for each row
    for $j \in \{1, 2, \ldots, |T_B|\}$ **do** ▷ for each column
      $mis \leftarrow T_A.char(i) \neq T_B.char(j)$
      $arr[i][j] \leftarrow \min \begin{cases} arr[T_A.par(i)][j] + 1, \\ arr[i][T_B.par(j)] + 1, \\ arr[T_A.par(i)][T_B.par(j)] + mis \end{cases}$
    end for
  end for
  $s_A \leftarrow 0, s_B \leftarrow 0$ ▷ evaluate (1)
  for $i \in T_A.terminalStates$ **do**
    $s_A \mathrel{+}= \min_{j \in T_B.terminalSt} \{arr[i][j]\} \cdot T_A.count[i]$
  end for
  for $j \in T_B.terminalStates$ **do**
    $s_B \mathrel{+}= \min_{i \in T_A.terminalSt} \{arr[i][j]\} \cdot T_B.count[j]$
  end for
  return $\frac{1}{2}\left(\frac{s_A}{|R_A|} + \frac{s_B}{|R_B|}\right)$ ▷ evaluate (2)
end function

[1] Sometimes term *similarity search* is used.

The complete pseudocode of the method is presented in Alg. 1. An example of evaluation on selected data is in Sect. 3.2.

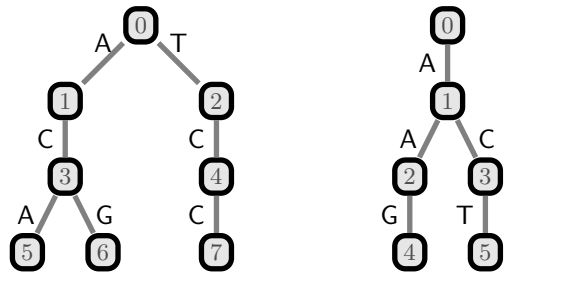
3.1. Notes about memory and time requirements

In Alg. 1 we allocate for simplicity an array that is long as product of number of nodes of two tries. This is actually not necessary. One dimension can have size of read length l if we traverse one of the tries similarly to dictionary search approach in [10]. This is applicable to dimension indexed with i in Alg. 1. For a node i we need to know row of table that belongs to its parent. Once we get to a leaf we can backtrack and reuse the memory used for current read. Also we need some additional memory for each leaf of trie T_B to store minimum used in calculation of (1).

In one dimension we need size l, in other we need the size of trie. Therefore the required space complexity can be reduced from $\Theta(|T_A||T_B|)$ to $\Theta(l \cdot \min\{|T_A|, |T_B|\})$. The time complexity remains in $\Theta(|T_A||T_B|)$. Our motivation to improve runtime of [9] however does not allow us to easily use approaches, in which some entries of dynamic programming table can be skipped.

3.2. Example

Consider two read bags $\{\mathsf{ACA}, \mathsf{ACG}, \mathsf{TCC}, \mathsf{TCC}\}$ and $\{\mathsf{AAG}, \mathsf{ACT}\}$. Their graphical representation as tries as well as internal representation in form of arrays are depicted in Fig. 1.



Node	0	1	2	3	4	5	6	7
parent	∅	0	0	1	2	3	3	4
char	∅	A	T	C	C	A	G	C
count	0	0	0	0	0	1	1	2

Node	0	1	2	3	4	5
parent	∅	0	1	1	2	3
char	∅	A	A	C	G	T
count	0	0	0	0	1	1

Fig. 1. Representation of read bags $\{\mathsf{ACA}, \mathsf{ACG}, \mathsf{TCC}, \mathsf{TCC}\}$ and $\{\mathsf{AAG}, \mathsf{ACT}\}$ as trie structures.

The tries are used to fill the dynamic programming table. In case of tries from Fig. 1 we obtain table.

	0	1	2	3	4	5
0	0	1	2	2	3	3
1	1	0	1	1	2	2
2	1	1	2	2	3	2
3	2	1	1	0	2	1
4	2	2	2	1	3	2
5	3	2	1	1	**2**	**1**
6	3	2	2	1	**1**	**1**
7	3	3	3	2	**3**	**2**

Using the values above we quantify the distance as

$$\frac{1}{2}\left(\frac{1\cdot 1+1\cdot 1+2\cdot 2}{4}+\frac{1\cdot 1+1\cdot 1}{2}\right)=\frac{5}{4}.$$

4. Experimental Evaluation

In this section we provide experimental evaluation of method proposed in Sect. 3. Our main evaluation criterion is runtime. We evaluated the algorithm for choices of read length $l \in \{8, 10, 12, \ldots, 30\}$ and for choices of read bag sizes $|R_A| = |R_B| \in \{100, 200, 300, \ldots, 2500\}$. The reads were sampled from DNA sequences of viruses with accession AM712239 and AM712239 that were downloaded from ENA repository. Both are available at `http://www.ebi.ac.uk/ena/data/view/<accession>`. We repeated each experiment 10 times and averaged the results for each choice of l and read bag size.

Implementation of both approaches was done in Java programming language with default setting. The experiments were performed on Linux system running on Intel Xenon E5-2630 v4 with 32 GB of RAM allocated. The implementations were single-threaded with maximum of shared code.

We compared results of Alg. 1 with the straightforward evaluation of (2) using standard Wagner-Fischer dynamic programming algorithm. Time needed to build trie is included in runtime of Alg. 1, because it is neglectable (approximately 0.1 %) compared to time that is needed to calculate distance (2).

The main experimental results are shown in Fig. 2 and Fig. 3. Figure 2 shows dependency of relative speedup w.r.t. reference on read length l. From the figure we see that for very short reads Alg. 1 is up to 3 times faster than the reference approach. However for longer reads the improvement is smaller. Figure 3 shows dependency of relative speedup of Alg. 1 on number of reads in each read bag.

Figure 4 shows overall dependence of relative speedup on read length and size of read bags. We see that the overall speedup is higher for short reads and bigger read bags. This has natural explanation in structure of the tries. They are able to compress shared prefix of the reads. Reads, unlike En-

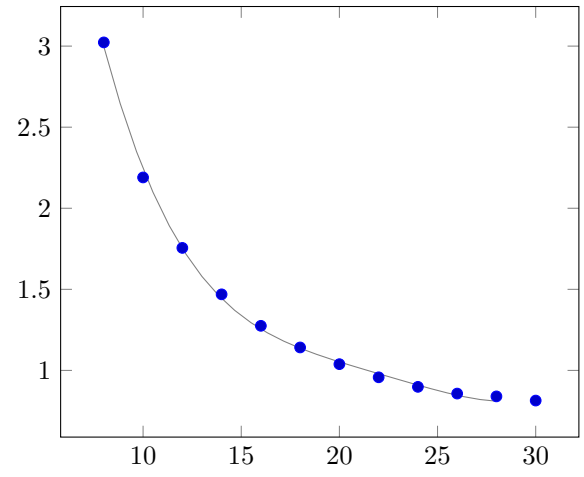


Fig. 2. Relative speedup w.r.t. reference on read length l. Values are averaged over choices of read bag size.

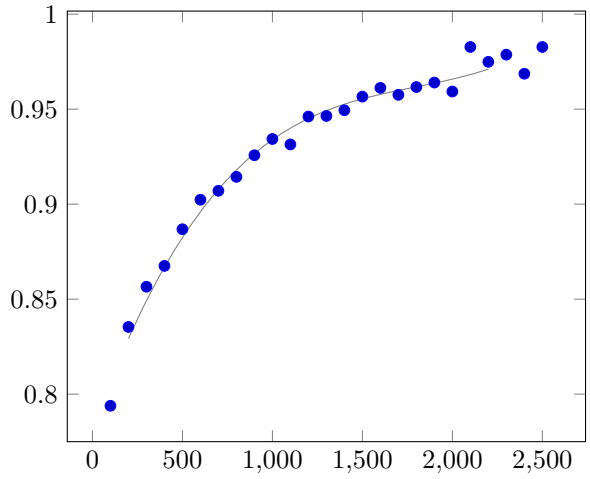


Fig. 3. Relative speedup w.r.t. reference on size of read bags. Values are averaged over choices of read length l.

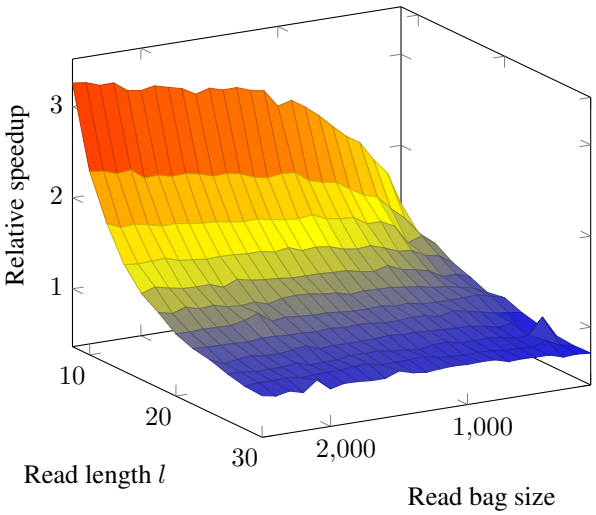


Fig. 4. Relative speedup w.r.t. reference on size of read bags and read length l.

glish words, are mostly random strings despite they are sampled from the same source. Therefore the common prefixes are less common than in case of dictionary search. Therefore a trie is able to avoid redundant work on approximately $\log_{|\Sigma|} |R_A|$ nucleotides. For $|R_A| = 2500$ and $|\Sigma| = 4$ this number is approximately 6. Therefore two reads are not likely to share more than 6 initial nucleotides. For longer reads, this shared prefix does not save much work and the savings are overtaken by lower memory locality.

5. Conclusion

We have seen a method that allows evaluation of Monge-Elkan distance for short reads faster than a natural approach by directly exploiting (2). This speedup is achieved by avoiding duplicate work that is caused by common prefixes of reads. The proposed method uses trie for exploring common prefixes in a way that is similar to approach [8] for similarity joins.

Because of the nature of data and application purposes the speedups are not that high compared to problem of dictionary search. However for very short reads (around 10) the proposed method gives speedups around 3. Current sequencing technologies usually produce longer reads, but DNA microarrays are used for probing for very short reads. Those techniques test a DNA sequence for existence of short fixed patterns, named probes. For longer reads (produced for example by Illumina technology) the proposed approach is not suitable and should be replaced for example by methods that do not evaluate exact value of (2), but only approximate it.

Acknowledgements

Research described in the paper was supervised by Prof. F. Železný, FEE CTU in Prague. This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS17/189/OHK3/3T/13. Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated. Author thanks to the reviewers for their helpful comments.

References

[1] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959.

[2] Jianhua Feng, Jiannan Wang, and Guoliang Li. Trie-join: A trie-based method for efficient string similarity joins. *The VLDB Journal*, 21(4):437–461, August 2012.

[3] Karam Gouda and Metwally Rashad. Prejoin: An efficient trie-based string similarity join algorithm. In *Informatics and Systems (INFOS),*

2012 8th International Conference on, pages DE–37. IEEE, 2012.

[4] David Hernandez and et al. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome research*, 18(5):802–809, 2008.

[5] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics doklady*, 10(8), 1966.

[6] Alvaro E Monge and Charles P Elkan. The webfind tool for finding scientific papers over the worldwide web. In *Proceedings of the 3rd International Congress on Computer Science Research*, 1996.

[7] Donald R. Morrison. Patricia — practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

[8] Astrid Rheinländer, Martin Knobloch, Nicky Hochmuth, and Ulf Leser. *Prefix Tree Indexing for Similarity Search and Similarity Joins on Genomic Data*, pages 519–536. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[9] Petr Ryšavý and Filip Železný. *Estimating Sequence Similarity from Read Sets for Clustering Sequencing Data*, pages 204–214. Springer International Publishing, Cham, 2016.

[10] H. Shang and T. H. Merrettal. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):540–547, Aug 1996.

[11] Jared T. Simpson and et al. Abyss: a parallel assembler for short read sequence data. *Genome research*, 9(6):1117–1123, 2009.

[12] Esko Ukkonen. Finding approximate patterns in strings. *Journal of algorithms*, 6(1):132–137, 1985.

[13] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

[14] Filip Železný, Karel Jalovec, and Jakub Tolar. Learning meets sequencing: a generality framework for read-sets. In *ILP 2014: 24th Int. Conf. on Inductive Logic Programming, Late-Breaking Papers*, 2014.

[15] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.