ACCURATE LONG READ MAPPING USING ENHANCED SUFFIX ARRAYS

Michaël Vyverman
Department of Applied Mathematics and Computer Science, Ghent University, Krijgslaan 281-S9, B-9000 Ghent, Belgium
Michael.Vyverman@UGent.be

Joachim De Schrijver, Wim Van Criekinge
Department of Molecular Biotechnology, Ghent University, Coupure Links 653, B-9000 Ghent, Belgium
Joachim.DeSchrijver@UGent.be, Wim.VanCriekinge@UGent.be

Peter Dawyndt, Veerle Fack
Department of Applied Mathematics and Computer Science, Ghent University, Krijgslaan 281-S9, B-9000 Ghent, Belgium
Peter.Dawyndt@UGent.be, Veerle.Fack@UGent.be

Keywords: inexact string matching, DNA sequencing, read mapping, enhanced suffix arrays

Abstract: With the rise of high throughput sequencing, new programs have been developed for dealing with the alignment of a huge amount of short read data to reference genomes. Recent developments in sequencing technology allow longer reads, but the mappers for short reads are not suited for reads of several hundreds of base pairs. We propose an algorithm for mapping longer reads, which is based on chaining maximal exact matches and uses heuristics and the Needleman-Wunsch algorithm to bridge the gaps. To compute maximal exact matches we use a specialized index structure, called enhanced suffix array. The proposed algorithm is very accurate and can handle large reads with mutations and long insertions and deletions.

1 INTRODUCTION

With the rise of the second generation sequencing technology the size of the sequencing data has increased dramatically. New applications and new challenges have arisen, including the need for new algorithms and data structures to handle the increased amount of data and new error models. The efficient mapping of sequencing reads to a reference genome is one of the most important challenges of the new sequencing technology, especially when considering the reassembly of genomes and its application in the prestiguous *1000 Genomes Project*.

Typical features of the new sequencing technology are the small read length and new types of sequencing errors. In recent years, many mapping programs have been developed for handling short read lengths. But, while sequencing technology advances, the read length of second generation sequencing technology is increasing and third generation sequencing promises read lengths of more than 1kbp. Moreover, most mapping algorithms for short reads focus on mutations as being the main error, while the longer 454 reads contain mostly insertions and deletions.

In this paper we focus on mapping reads of several hundreds of base pairs to a medium sized reference sequence, where the reads may contain a small number of long insertions and deletions, as well as mutations. The proposed method is based on chaining together maximal exact matches (MEMs) using heuristics for speeding up the search. In order to obtain the MEMs between query and reference sequence, we store the reference sequence in a specialized data structure, the so-called enhanced suffix array (ESA) (Abouelhoda et al., 2004). The MEMs are used as anchors in the alignment and are combined or chained in a full local alignment using a combination of the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970) for global alignment together with some heuristics.

Related Work. Many fast and accurate short-read mappers exist today. *SOAP* (Li et al., 2008), *Bowtie* (Langmead et al., 2009) and *BWA* (Li and Durbin, 2009) all perform well when used for short reads with few mismatches. The upper bounds for the read length are 200bp for *BWA* and 1024bp for *Bowtie* and *SOAP*. Hoffmann et al. introduced a new short sequence mapper *segemehl* and compared its performance with *SOAP*, *BWA* and *Bowtie* (Hoffmann et al., 2009). Their results show that most mapping programs are not capable of forming a correct alignment when 4 or more differences between query and reference sequence are present. Recently a fast and accurate long read mapper *BWA-SW* using the Burrows-Wheeler transform was introduced (Li and Durbin, 2010).

Also mapping programs based on maximal exact matches have already been investigated. For example, *AVID* (Bray et al., 2003) is a global alignment program that uses MEMs as anchors for a global alignment. *AVID* further uses recursion and in a final step the Needleman-Wunsch algorithm is used in order to fill the gaps between the MEMs. MUMmer (Kurtz et al., 2004) is another alignment program that uses a suffix tree-based seed and extension algorithm. MUMmer uses maximal unique matches (MUMs) instead of MEMs.

2 PRELIMINARIES

By $\Sigma = \{A, C, G, T\}$ we denote the alphabet of nucleotides. Let S be a reference sequence over the alphabet Σ and let Q be a set of query sequences with $q \in Q$. The *inexact string matching problem* is defined as finding an optimal local alignment between the query sequences and the reference sequence S. The queries can contain mismatches and gaps. The differences can be the result of genetic differences, sequencing errors or other sources; however, the source of the differences is not important for our purposes and the gaps will be referred to as *mutations*, *insertions* and *deletions*. The solution of the stated problem gives a starting position, followed by a list of the positions of the insertions, deletions and mutations along with their length if necessary. The *edit distance* between S and a query q is defined as the number of 'errors' (insertions, deletions or mutations) of the optimal local alignment found.

A *maximal exact match* (MEM) can intuitively be seen as an exact match between substrings of two sequences that is no substring of another exact matching substring between the two sequences. In this context, a maximal exact match between the reference sequence S and the query sequence q is a triple (q_s, ℓ, P), where q_s is a position in q corresponding to the start of the match in q, ℓ is the length of the match and P is a list of index positions p_i in S that correspond to starting positions of the match in S. When the list P contains only one index position p we can also denote the MEM as (q_s, ℓ, p). The substrings corresponding to $[q_s, q_s + \ell], [p_1, p_1 + \ell], \ldots, [p_{|P|}, p_{|P|} + \ell]$ are exact matches. The matches are maximal in the sense that no exactly matching substring in S can be found for the substrings corresponding to $[q_s - 1, \ell]$ and $[q_s, \ell + 1]$. The definition of maximal exact matches here is biased towards the query sequence. The reason for this bias is that maximal exact matches will be used to traverse the query sequence from left to right, allowing the same substring in q to be encountered more than once.

A *suffix tree* (Gusfield, 1997) for a string S (denoted by $\mathcal{T}$) is a tree-like data structure that indexes S, by forming a one-to-one correspondence between the suffixes of the string and the leaves of the tree. The main assets of the suffix tree are its construction cost, which is linear in the length of the string, and its pattern matching cost, which is linear in the length of the pattern.

Because many implementations and variants of suffix trees exist, it is important to note that $\mathcal{T}$ is a suffix tree with suffix links. In our implementation we use a variant of the suffix tree, a so-called *enhanced suffix array* (ESA) (Abouelhoda et al., 2004), although the idea behind the algorithm intuitively uses a suffix tree. We will use the general term *index structure* to refer to variants of both suffix trees and suffix arrays.

3 ALGORITHM

3.1 Overview

The proposed algorithm first builds an enhanced suffix array (ESA) for the reference sequence S and then processes the query file one query at a time. Note that the ESA data structure for S can be reused for all the queries and in later projects. The ESA index is first used to calculate maximal exact matches (MEMs) between the reference sequence S and a query. Afterwards, the obtained MEMs are filtered (using some heuristics). After filtering, the MEMs roughly corresponding to the best alignment position are joined together. The chaining is done by traversing the list of filtered MEMs from left to right, relative to their starting positions in the query sequence. The actual joining process uses the Needleman-Wunsch global alignment algorithm.

The main idea behind the algorithm is based on first pinpointing the exact position of the query in the reference genome corresponding to the optimal alignment. Then the MEMs can be seen as a sort of anchors or seeds in the alignment. The chaining idea is based on dynamic programming, where the matrix rows correspond to the query, while the colums correspond to the reference sequence and the MEMs are diagonals in the matrix. The chaining algorithm tries to combine diagonals that score well in order to find an optimal alginment.

Algorithm 1 Main Algorithm

```
Require: $S$, $\mathcal{T}$, $q \in Q$, expected edit distance $k$
Ensure: an optimal local alignment of $q$ in $S$
 1: $mems \leftarrow$ mems($S$,$q$,$\mathcal{T}$)
 2: $mems \leftarrow$ filter($mems$, $q$, $k$)
 3: $mem \leftarrow mems$.get(1)
 4: if $mem.q_s > 0$ then
 5:     $sol \leftarrow$ dynProgBegin($S$,$q$,$mem$,$k$)
 6: else
 7:     $sol \leftarrow mem.P$ {starting pos. of alignment in $S$}
 8: $i \leftarrow 2$
 9: while ($i \leq |mems|$) do
10:     $j \leftarrow i$
11:     while $j \leq |mems| \wedge$ distance between $mem$ and
        $mems$.get($j$) is descending do
12:         $j \leftarrow j + 1$
13:     $newMem \leftarrow mems$.get($j - 1$)
14:     $sol \leftarrow sol +$ findGap($S$,$q$,$mem$,$newMem$)
15:     $mem \leftarrow newMem$
16:     $i \leftarrow j$
17: if $mem.q_s + \ell < |q|$ then
18:     $sol \leftarrow sol +$ dynProgEnd($S$,$q$,$mem$,$k$)
19: return $sol$
```

3.2 Main Algorithm

Algorithm 1 requires as input the reference sequence S, a query q, the index structure $\mathcal{T}$ of $S\$$, and the expected edit distance k. The expected edit distance k is an important parameter but can be an approximation. The algorithm will try to find an optimal alignment of q in a substring of S of size $|q| + 2k$.

The matching process starts by finding the list of all MEMs between S and q (line 1). In the next step (line 2), the list is filtered so that only the most promising MEMs remain. For every MEM (q_s, ℓ, P) remaining after the filtering, it holds that P contains only one element. In most cases, the first MEM is part of the best alignment, but even in the worst case, the filtering process guarantees that it is part of a suboptimal alignment that is located within k positions of the optimal alignment.

During the main loop (lines 9-16), the list of filtered MEMs is traversed. For every current MEM, corresponding to an interval in q and an interval in S, the closest MEM to the right of the current match is searched. More details on the definition of 'closest' MEM and the steps in the inner loop (lines 11-12) are given in the next paragraphs. Once the closest MEM is found, the gap between the two matches is bridged (line 14), using a mix of heuristics and the Needleman-Wunsch global alignment algorithm.

It is possible that, due to errors in the first and/or last characters of q, the alignment obtained during the main loop does not cover the entire query. The functions dynProgBegin() (line 5) and dynProgEnd() (line 18) are used to find those parts of the alignment.

3.3 Calculating the MEMs

The maximal exact matches between S and q are found using an adaptation of the matching statistics algorithm described in (Gusfield, 1997). The main idea behind the algorithm consists of matching every suffix of q to the suffix tree of S. After a mismatch is found for one suffix, a suffix link is used in order to speed up the search for the next suffix. Using some tricks, the algorithm runs in linear time complexity $O(|q|)$. The main difference between their algorithm and our approach is that the reference text S and query (or pattern) q are switched. Normally, it would be more interesting to build the index structure for the shortest sequence (i.e. q), but in our case building an index structure once for the reference sequence is more interesting than building an index structure for every query, since the sum of the lengths of the queries is much larger than the length of S. The benefit of this approach is that the MEMs are sorted in increasing starting position in q. Also, the list P, corresponding to the positions of a match in S, is sorted in increasing order. In order to suppress the output of the algorithm, a minimal length depending on k and $|q|$ is induced upon the MEMs.

3.4 Filtering the MEMs

The list of MEMs and their lists P of positions in S can be large. Filtering the list of MEMs using fast heuristics has the advantage of speeding up the main loop in Algorithm 1, but it also allows to find the first MEM (with the smallest q_s) that is contained in an optimal local alignment.

First the list of MEMs is traversed in order to find a match with maximal length, which is assumed to be in the optimal alignment and is referred to as the anchor of the alignment. Searching for a longest match allows some error on the choice of parameter k. Next, the list is traversed again and filtered, keeping only MEMs that 'have potential' and that contain a position $p \in P$ that is 'close to' the anchor.

A MEM (q_s, ℓ, P) is said to 'have potential' if the following condition holds:

$$(\ell \geq \frac{|q|}{k}) \vee (|P| = 1) \vee (\ell = \ell'),$$

where ℓ' is the length of the anchor. This means that the heuristic allows long matches, matches that are unique in S and matches with the same length of the anchor. The last of these conditions is important when dealing with short reads, highly repetitive regions or a wrong estimate of k.

The condition that decides whether a potential match is added to the filtered list, traverses the list P and searches the first position p such that the relative position of the current match to the anchor is located within a window $[-k, +k]$ of the position it would have if there were no errors or differences in the alignment.

Experiments show that the combination of the above heuristics filters almost always all MEMs not corresponding to an optimal alignment related to the longest MEM, while keeping most of the MEMs that do form an optimal alignment around the longest MEM. We believe that repeating the filtering step, but using a different reference MEM instead of the longest match, allows to find other alignments.

3.5 Finding the 'Closest' MEM

The distance between two MEMs (q_s, ℓ, p) and $(q_{s'}, \ell, p')$ is defined as follows. First, distances in q and S are determined as $qDist := q_s + \ell - q_{s'}$ and $refDist := p + \ell - p'$. If both distances are positive, which is the most common situation, the distance between the two MEMs is the maximum of the two distances *qDist* and *refDist*. If both are negative, resulting in overlapping MEMs, the resulting distance is $|\,|refDist| - |qDist|\,|$. If one is positive and the other negative then the distance is defined as the sum of the positive distance and the absolute value of the negative distance.

If at least one of the differences is negative (*qDist* or *refDist*), then the shortest distance between the MEMs is a single insertion or deletion (according to the chosen score function). The case of two positive differences is explained in the next section.

Using this definition and the ordering on the list of MEMs from the first step, one can see that the distance between the two MEMs in general only descends at first, reaches a minimum and then starts increasing. This means that during the main loop of the algorithm every MEM is checked at most twice.

3.6 Closing the Gaps

Gaps and mismatches can occur between the beginning of the query and the first MEM in the list, between two MEMs in the list and between the last MEM and the end of the query.

The first and last gaps are bridged by using a variant of the Needleman-Wunsch global alignment algorithm, usually referred to as semi-global alignment. The first variant allows gaps at the beginning of S that are not penalized and the second variant allows for free gaps at the end of S.

In order to find the gaps between two MEMs, the distance and the differences from the previous paragraph are used. First, the algorithm checks if a single insertion, deletion or mutation fills the gap. Otherwise, a global alignment is performed between the right bounds of the first MEM and the left bounds of the second MEM.

The Needleman-Wunsch algorithm was always executed with a score $+2$ for matches, -1 for mismatches and an affine gap penalty with an opening gap penalty of -2 and an extension penalty of -1. Other scores can be used, resulting in a different alignment, especially at the ends of the query.

3.7 Implementation and Complexity

We implemented the above algorithm in Java. An enhanced suffix array (ESA) was used instead of a standard suffix tree because an ESA has a lower memory footprint, has a modular design and maintains the full functionality of the suffix tree. The ESA (Abouelhoda et al., 2004) contains four arrays: the suffix array, the longest common prefix (lcp) table, the child table and the suffix link table. The suffix array was constructed using the difference cover algorithm of Karkkainen and Sanders (Kärkkäinen and Sanders, 2003). The lcp table was constructed using the algorithm of Kasai et al. (Kasai et al., 2001). The child table was constructed using the algorithms described in (Abouelhoda et al., 2004) and finally the suffix link table was constructed using Maaß algorithm (Maaß, 2007).

The memory footprint of the algorithm mainly depends on the size of the ESA and storage of the reference sequences and query sequences. The implementation of the ESA without memory optimization takes about 20 bytes per character, which results in an index structure of about 100MB for a sequence like *E. coli*. Since the query sequences are in general much shorter than the reference sequence and MEMs are not too abundant, the memory needed for storing the MEMs is low.

The theoretical time complexity of the algorithm depends on the construction of the index structure, the search for MEMs, and the filtering and chaining of the MEMs using the Needleman-Wunsch algorithm. For one query q of length m and a reference sequence S of size n the asymptotic time complexity can be calculated as follows. Constructing the ESA can be done in $O(n)$ time (Abouelhoda et al., 2004), while calculating the MEMs can be done in $O(m)$ time (Gusfield,

Table 1: Alignment of the simulated data using our algorithm and *Bowtie*. The row '# of different alignments' gives the number of queries where our algorithm found an optimal alignment that differed from the alignment suggested by simulating the errors.

	Our algorithm	*Bowtie*
time (sec.)	7.5	264
# of correct alignments	75 381	24 252
# of different alignments	1 835	
# of failed alignments	2 150	55 123
percentage correct	95%	31%

1997). Finding the positions P for every MEM can be done in constant time using an ESA. Filtering the MEMs can be done in $O(\sum P)$, where the sum runs over the number of found MEMs, which is bounded by m. In theory, if the lists P are large (meaning a lot of repeats or small maximal exact matches), this sum can become very large. However, in practice the heuristics speed this up considerably. The main loop of Algorithm 1 can be executed in $O(m)$ time plus the time used for the executions of the Needleman-Wunsch algorithm. The worst case scenario is when only one small MEM is found, which translates in a time complexity of $O((m+2k)^2)$, which is the time complexity of a Needleman-Wunsch alignment, when the position of q in S is known. When many or long MEMs are found and not filtered out, the main loop is very fast. Currently the bottleneck of the algorithm is the calculation of the maximal exact matches.

4 RESULTS

4.1 Preliminary Results

We tested our algorithm on simulated BRCA1 data. BRCA1 is a gene known to be involved in the development of breast cancer when mutated (Friedenson, 2007) and is routineously screened in older women. The diagnostic community is currently implementing next-generation sequencing to carry out these screenings, hence BRCA1 data was used to test and validate the algorithm.

A reference sequence of 80kbp was used together with a query file containing about 80000 queries ranging in size from 8bp to 1021bp with an average length of about 600bp. The total number of insertions, deletions and mutations for one query was bounded by 4 errors of each type, while on average the total number of errors was 2.5. The longest indels were 10bp long and the average edit distance was 5.73.

The simulated queries were aligned with our algorithm and *Bowtie* (Langmead et al., 2009), where *Bowtie* was executed with the default parameters. The results of the alignments are shown in Table 1. This includes the running time in seconds, as well as the number of correct and failed alignments. Note that in some cases our algorithm finds an alignment that is different from what would be suggested by the simulation of the errors, but has the same edit distance; these cases are also shown in the table.

The tests show that our algorithm is both faster and more accurate than *Bowtie*. However, note that *Bowtie* was not optimized for performance, which could explain its low speed. The large amount of failed alignments for *Bowtie* can be explained by the large length of the queries and the high edit distance and/or large insertions and deletions.

The results show that our algorithm was able to find an optimal alignment for almost all the queries (95% of them). The set of failed alignments contains 129 queries with a larger edit distance and 2021 queries with a smaller edit distance (than the alignment given by the simulation). Some of the alignments with smaller edit distance are due to the scoring in the Needleman-Wunsch algorithm. The failed alignments are mostly caused by using wrong MEMs in the alignment, which stem from too rigorous filtering of the MEMs or the greedily defined distance between MEMs. However, even if part of the alignment failed for a given query, the 'true position' of the query was found in all but 6 cases.

4.2 Current and Future Work

Currently we are testing our algorithm for datasets with longer reads and more erroneous reads. We also work on comparing its results with other long read aligning programs, such as *BWA-SW* (Li and Durbin, 2010), *MUMmer* (Kurtz et al., 2004) and *RazerS* (Weese et al., 2009).

In this version of the algorithm, only one possible best local alignment is returned. This is consistent with the default option of many widely used alignment programs today. In future versions of the algorithm, an option could be included allowing more alignments to be returned. There can be some issues with small reads and reference sequences containing a lot of long repetitive regions, because a good first anchor MEM cannot be found in those circumstances. However, since the latter is not regularly screened by sequencing, this present no big problems in practice. When screening repeat regions, both the biology as the informatics need some customization, which is not the scope of this manuscript.

In spite of its linear memory usage, the index structure used in the current implementation still has a relatively large memory footprint (i.e. 20 bytes per character). According to Abouelhoda et al. (Abouelhoda et al., 2004), the memory footprint can be scaled down to 8 bytes per character. Moreover, the main bottleneck in speed is finding the MEMs. Recently, Khan et al. (Khan et al., 2009) used a sparse suffix array to compute MEMs. Also, many mapping programs use an FM-index instead of suffix trees to improve the memory requirements. In future versions, our algorithm could implement a more memory-friendly index structure. A possibility to increase accuracy is changing the anchors or seeds from MEMs to maximal unique matches as in *MUMmer* (Kurtz et al., 2004) or using spaced or inexact matches.

5 CONCLUSION

We presented an algorithm for mapping large reads to a reference genome which is fast and accurate in finding an optimal local alignment. The algorithm is easy to understand and has few parameters. Our first results show that the algorithm is able to map reads with insertions, deletions and mutations and with a length of several hundreds of base pairs successfully to a reference sequence.

The algorithm is more accurate given longer queries with lower error rates. The main algorithm takes several parameters which can be tuned. These are: the expected edit distance and the cost parameters for the Needleman-Wunsch algorithm. The expected edit distance can be approximated and allows some deviation from the real edit distance. Better results are obtained with an overestimate than with an underestimate. The algorithm is devised in such a way that an alignment is always found even if it is suboptimal. Even if reads contain insertions, deletions or mutations in the first or last parts of the query, our algorithm was able to find an optimal alignment.

ACKNOWLEDGEMENTS

The research of MV is funded by a grant from the N2N-MRP at Ghent University.

REFERENCES

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86.

Bray, N., Dubchak, I., and Patcher, L. (2003). AVID: a global alignment program. *Genome Research*, 13:97–102.

Friedenson, B. (2007). The BRCA1/2 pathway prevents hematologic cancers in addition to breast and ovarian cancers. *BMC Cancer*, 7:152.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences*. Cambridge university press, 32 Avenue of the Americas, New York, NY 10013-2473, USA, 11th edition.

Hoffmann, S., Otto, C., Kurtz, S., Sharma, C., Khaitovich, P., Vogel, J., Stadler, P., and Hackermüller, J. (2009). Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS Computational Biology*, 9:e1000502.

Kärkkäinen, J. and Sanders, P. (2003). Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata Languages and Programming*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955. Springer-Verlag.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., and Park, K. (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM 01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer-Verlag.

Khan, Z., Bloom, J., Kruglyak, L., and Singh, M. (2009). A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 13:1609–1616.

Kurtz, S., Phillippy, A., Delcher, A., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. (2004). Versatile and open software for comparing large genomes. *Genome Biology*, 5:R12.

Langmead, B., Trapnell, C., Pop, M., and Salzberg, S. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10:R25.

Li, H. and Durbin, R. (2009). Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25:1754–1760.

Li, H. and Durbin, R. (2010). Fast and accurate long read alignment with Burrows-Wheeler transform. *Bioinformatics*, 5:589–595.

Li, R., Li, Y., Kristiansen, K., and Wang, J. (2008). SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24:713–714.

Maaß, M. (2007). Computing suffix links for suffix trees and arrays. *Information Processing Letters*, 101:250–254.

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. *Journal of Molecular Biology*, 48(3):443–453.

Weese, D., Emde, A.-K., Rausch, T., Döring, A., and Reinert, K. (2009). RazerS – fast read mapping with sensitivity control. *Genome Research*, 19:1646–1654.