Fast accurate sequence alignment using Maximum Exact Matches

Arash Bayat   Aleksandar Ignjatovic
Bruno Gaeta   Sri Parameswaran

University of New South Wales, Australia
{a.bayat, bgaeta}@unsw.edu.au, {ignjat, sridevan}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201701
Feb 2017

Abstract

Sequence alignment is a central technique in biological sequence analysis, and dynamic programming is widely used to perform an optimal alignment of two sequences. While efficient, dynamic programming is still costly in terms of time and memory when aligning very large sequences. We describe MEM-Align, an optimal alignment algorithm that focuses on Maximal Exact Matches (MEMs) between two sequences, instead of processing every symbol individually. In its original definition, MEM-Align is guaranteed to find the optimal alignment but its execution time is not manageable unless optimisations are applied that decrease its accuracy. However it is possible to configure these optimisations to balance speed and accuracy. The resulting algorithm outperforms existing solutions such as GeneMyer and Ukkonen. MEM-Align can replace edit distance-based aligners or provide a faster alternative to Smith-Waterman alignment for most of their applications including in the final stage of short read mapping.MEM-Align is publicly available at https://sites.google.com/site/memalignv1.
Supplementary data section is attached.

1 Introduction

Biological sequence alignment is a fundamental problem in bioinformatics that underlies many biological analyses. Dynamic programming is widely used to compute the optimal alignment between two sequences [1]. Although several variations of the optimal alignment algorithm have been proposed including Needleman-Wunsch [2] and Smith-Waterman [3], they all rely on a dynamic programming strategy in which input sequences are laid on the vertical and the horizontal axes of a dynamic programming table.

Since the size of the dynamic programming table is not manageable for lengthy sequences (eg an entire genome, or a database of genes), heuristic methods have been introduced that search for subsequences of the query sequence in a large target sequence to identify similar regions to be processed using an optimal alignment algorithm. This search operation cannot guarantee that the optimally-aligning region is always identified. Database search tools such as BLAST [4], DNA read mappers such as BWA [5] and genome aligners such as MUMmer [6] are a few examples of heuristic alignment, all of which rely on a form of dynamic programming-based optimal alignment in their late processing stage.

Since the Smith-Waterman algorithm is widely used by many bioinformatic applications [7–10], it has been accelerated using a range of strategies including vectorized instructions of the target processor [11–14], graphics processors [15] as well as hardware accelerators [16, 17]. However this is often insufficient or impractical and many applications substitute optimal dynamic programming alignment with a fast edit-distance based alignment method where the goal of the alignment is to minimise the number of differences (edits) but not to produce the optimal alignment with maximum score. These fast approximate alignments are likely to produce an optimal alignment if the number of edits is quite low. GEM [18] and SNAP [19] are two examples which take advantage of the speed of the modified version of GeneMyer [20] and the Ukkonen algorithm [21] respectively.

In this paper, we present a novel alignment algorithm using dynamic programming to replace traditional optimal alignment that looks into the problem from a different perspective. Instead of aligning the sequences with the granularity of symbols, our proposed algorithm (MEM-Align) processes Maximal Exact Matches (MEMs) that exists between two sequences. An exact match is a common subsequence of two sequences and is maximal when it is not a subsequence of a larger exact match.

MEM-align, in its original definition, is guaranteed to find the optimal alignment. However, its execution time is not manageable unless a series of optimisations is applied. These optimisations come at the cost of slight inaccuracies, and the resulting alignment is not guaranteed to be optimal. However, since the introduced optimisations are tuneable, it is possible to configure them to balance the speed and accuracy of the produced alignment. Our results demonstrate significant speed ups at the cost of a minimal drop in accuracy.

Given a pair of sequences, MEM-Align extracts and sorts all possible MEMs, and then processes them in order to align them. The contributions in this paper include a fast bit-vector method to extract MEMs, a novel alignment algorithm based on MEMs as well as a series of optimisations that speed up both MEM extraction and the alignment process. MEM-Align is mainly designed to align nucleotide sequences but we provide guidelines on how to extend the algorithm to align protein sequences as well.

The rest of this paper is organised as follows. In Section 2 a formal definition of an optimal alignment is provided and in Section 3 a new alignment representation is discussed which is the basis for an alignment algorithm introduced in Section 4. Section 5 elaborates a fast method to extract MEMs and Section 6 describes a series of critical optimisations. Additional explanations, proofs and implementation details are provided in section 7. Section 8 presents experimental results, and Section 9 summarised future works.

2 Background

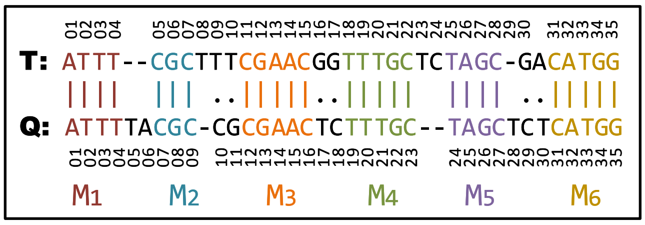
An alignment is a mapping between two sequences which pairs each symbol of one sequence with either a symbol of the other sequence or a gap. When the symbols of a pair are identical, the pair is called a match; otherwise, it could be a mismatch or a gap. Figure 2.1 is an example alignment between a target sequence T and a query sequence Q in which matches, mismatches and gaps are shown by "|", ".", "-" respectively.

The total number of gaps and mismatches in the alignment is called the edit-distance and is the number of edits needed to be applied to T in order to transform it into Q. Some alignment algorithms aim to minimise the edit-distance when aligning sequences. However, the optimal alignment is defined differently, as the alignment that maximises the alignment score. The alignment score is computed based on a scoring system, typically composed of four numbers: match score R_m; mismatch penalty P_x; gap open penalty P_o; and gap extend penalty P_g. While the R_m, P_x and P_e are applied for each individual match, mismatch, and gap respectively, P_o is applied for each group of continuous gaps once only. Given the number of matches N_m, mismatches N_x and gaps N_g as well as number of groups of continuous gaps N_o, in the gap-affine model, the alignment score AS is computed using Equation 2.1. Figure 2.2, illustrates the computation of the alignment score for the example alignment.

3 Approach

Each alignment can be represented as an ordered list of exact matches. There should be at least a gap or a mismatch to separate consecutive exact matches. No match should exist between consecutive exact matches as it would forms another exact match. Figure 2.3, is a list of six exact matches (M_1 to M_6) that form the alignment. Each exact match M_i is stored as triplet integer numbers: the beginning positions in T and Q (BT_i and BQ_i respectively) and its length (L_i). The ending positions in T and Q, are computed using Equation 2.2a and 2.2b respectively.

In order to compute the alignment score for a list of exact matches, we first compute the length of the subsequences between two exact matches (M_i and M_j) in T and Q using Equation 2.2c and 2.2d respectively. For example, consider M_2 and M_3 in Figure 2.1 where there are three symbols between them in T ($LT_3^2 = 3$) and only two symbols in Q ($LQ_3^2 = 2$). Since three symbols of T can not be mapped with two symbols of Q, at least one symbol of T should be



T: ATTT--CGCTTTCGAACGGTTTGCTCTAGC-GACATGG
Q: ATTTTACGC-CGCGAACTCTTTGC--TAGCTCTCATGG
M1
M2
M3
M4
M5
M6

Figure 2.1: An example alignment.

$$AS = (N_m \times R_m) - (N_x \times P_x) - (N_g \times P_g) - (N_o \times P_o) \tag{2.1}$$

$$Scoring \begin{cases} R_m = 2 \\ P_x = 3 \\ P_g = 1 \\ P_o = 4 \end{cases} \qquad Alignment \begin{cases} N_m = 27 \\ N_x = 6 \\ N_g = 5 \\ N_o = 3 \end{cases}$$
$$AS = (27 \times 2) - (6 \times 3) - (5 \times 1) - (3 \times 4) = 18$$

Figure 2.2: Computing the alignment score for the example alignment in Figure 2.1.

	M1	M2	M3	M4	M5	M6
BT:	1	5	11	18	25	31
BQ:	1	7	12	19	24	31
L :	4	3	5	6	4	5

Figure 2.3: List of exact matches of the example alignment in Figure 2.1.

$$ET_i = BT_i + L_i \tag{2.2a}$$
$$EQ_i = BQ_i + L_i \tag{2.2b}$$
$$LT_i^j = BT_i - ET_j - 1 \tag{2.2c}$$
$$LQ_i^j = BQ_i - EQ_j - 1 \tag{2.2d}$$
$$LD_i^j = LT_i^j - LT_i^j \tag{2.2e}$$
$$N_g^{j,i} = |LD_i^j| \tag{2.2f}$$
$$\hat{N}_x^{j,i} = \min(LT_i^j, LQ_i^j) \tag{2.2g}$$

$$N_m = \sum_{i=1}^{n-1} L_i \tag{2.3a}$$
$$N_x = \sum_{i=1}^{n-1} \hat{N}_x^{i,i+1} \tag{2.3b}$$
$$N_g = \sum_{i=1}^{n-1} N_g^{i,i+1} \tag{2.3c}$$
$$N_o = \sum_{i=1}^{n-1} \begin{cases} 1 & N_g^{i,i+1} \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.3d}$$

mapped to a gap that indicates a deletion. The absolute difference between LT_i^j and LQ_i^j represents the length of the gap between M_i and M_j which is computed using Equation 2.2f. Note that a negative LD_i^j (computed using Equation 2.2e) indicates an insertion and a positive LD_i^j indicates a deletion. The number of mismatches between M_i and M_j is then computed using Equation 2.2g. Finally, the total number of matches, mismatches and gaps as well as groups of continuous gaps for a set of exact matches that forms a valid alignment are computed using Equation 2.3a, 2.3b, 2.3c and 2.3d.

The placement of gaps and mismatches is not important for nucleotide sequences where the mismatch penalty is constant. As a consequence, if gaps exist between two exact matches, all gaps are considered to be continuous and attached to one of the exact matches to minimise the effect of the gap open penalty.

4 Alignment algorithm

In our example in Figure 2.1, M_1 to M_6 are not the only exact matches between T and Q as some others are shown in Figure 4.1. MEM-Align is designed to find a set of exact matches that forms an optimal alignment given a list of all MEMs. For simplicity, first an algorithm is described that takes the list of all exact matches; we then extend the algorithm to process only maximal exact matches which are much fewer in number. Each exact match is a subsequence of a maximal exact match; for example, if ACG is a maximal exact match then A, C, G, AC, and CG are other shorter exact matches to be considered.

4.1 Alignment using exact matches

Roughly speaking, dynamic programming is a method of finding a global solution from available local solutions. In an optimal alignment problem where each alignment ends in an exact match, the optimal alignment score $\hat{S}_i$ (computed using Equation 4.1d) for the alignment which ends in i^{th} exact match M_i could be considered as a global solution. Local solutions which contribute to the calculation of $\hat{S}_i$ are the optimal alignments ending at exact matches which appear before M_i in both T and Q. $\hat{F}_i^j$ (computed using Equation 4.1a) is true when M_j appears before M_i in both T and Q; and false otherwise. Since M_i can not be a part of the alignment ending at M_j where $\hat{F}_i^j = ture$, the local solution $\hat{S}_j$ can be computed independently from the global solution $\hat{S}_i$. As a result, a dynamic programming strategy is applicable to this problem. Note that the global and local solutions in dynamic programming discussed below are not to be understood as global and local alignments.

Given the list of all exact matches, the first algorithmic step is to sort the list of exact matches based on ending position in Q. The sorting guarantees that $\hat{S}_j$ is computed independently from $\hat{S}_i$ for $j < i$. The complexity of the sorting operation is $n\,log(n)$ on average if using quick sort where n is the number of exact matches. However, as the ending position in Q is a small number, the sorting can be optimized using the counting sort algorithm with complexity of $O(3n)$. The implementation of counting sort is elaborated in *Supplementary Data* Section 10.2. $\hat{S}_i^j$ (computed using Equation 4.1c) is a function which extends the optimal alignment ending at M_j ($\hat{S}_j$ as a local solution) with M_i,

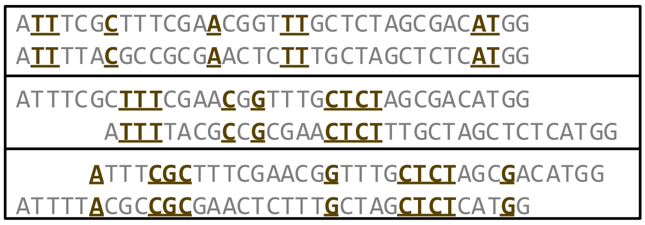

ATTTCGCTTTCGAACGGTTTGCTCTAGCGACATGG
ATTTTACGCCGCGAACTCTTTGCTAGCTCTCATGG
ATTTCGCTTTCGAACGGTTTGCTCTAGCGACATGG
ATTTTACGCCGCGAACTCTTTGCTAGCTCTCATGG
ATTTCGCTTTCGAACGGTTTGCTCTAGCGACATGG
ATTTTACGCCGCGAACTCTTTGCTAGCTCTCATGG

Figure 4.1: Alternative exact matches between the sequence pair in Figure 2.1

$$\hat{F}_i^j = \begin{cases} true & EQ_j < BQ_i \wedge ET_j < BT_i \\ flase & \text{otherwise} \end{cases} \tag{4.1a}$$

$$\hat{P}_i^j = (\hat{N}_x^{j,i} \times P_x) + (N_g^{j,i} \times P_g) + \begin{cases} P_o & N_g^{j,i} \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1b}$$

$$\hat{S}_i^j = \begin{cases} S_j + (L_i \times R_m) - \hat{P}_i^j & \hat{F}_i^j = true \\ 0 & \text{otherwise} \end{cases} \tag{4.1c}$$

$$\hat{S}_i = \max \begin{cases} \max\limits_{1 \leq j \leq i-1} \hat{S}_i^j \\ (L_i \times R_m) \end{cases} \tag{4.1d}$$

$$\hat{S} = \max_{1 \leq i \leq n} \hat{S}_i \tag{4.1e}$$

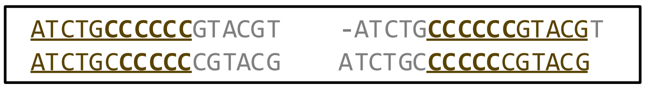


Figure 4.2: An example of two overlapping MEMs

where extension is possible ($\hat{F}_i^j = ture$). The extension adds to $\hat{S}_j$ the score for all matches in M_i ($L_i \times R_m$), and then subtracts the penalty for gaps and mismatches that separate M_j and M_i denoted by $\hat{P}_i^j$. $\hat{P}_i^j$ is computed using Equation 4.1b from numbers of mismatches and gaps existing between M_j and M_i.

The global solution ($\hat{S}_i$) is the maximum of all extended local solutions ($\hat{S}_i^j$). Since our algorithm is designed for local alignment, in Equation 4.1d, we also include the case where M_i is considered to be the first exact match in the alignment which results in the score of $L_i \times R_m$. In a local alignment, leading and trailing symbols can be excluded from the alignment if the exclusion results in a higher score; thus the optimal local alignment can begin and end with any exact matches. The optimal local alignment score for the entire T and Q ($\hat{S}$) is computed as the maximum score for the alignment ending at any of the exact matches using Equation 4.1e. *Supplementary Data* Section 10.3 explains how to modify this algorithm to produce the global alignment.

When computing P_i^j, for simplicity, the algorithm assumes that the whole area between M_i and M_j is composed of gaps and mismatches and that there are no matches in between. Although this assumption is not true for every M_j, it is always true for the M_j that leads to maximum $\hat{S}_i^j$ which overrules the effect of the assumption being incorrect for other M_j.

Now that the optimal alignment score is computed, the list of exact matches in the optimal alignment is obtained through a backtracking process. For each exact match M_i, the backtracking process requires $W(i)$ such that $M_{W(i)}$ maximizes the score for M_i, or more formally $S_i = S_i^j$. $W(i)$ is computed and stored during computation of S_i. If M_i is an exact match in the optimal alignment, then $M_{W(i)}$ is the immediate previous exact match of M_i in the optimal alignment. As a result, $i \leftarrow W(i)$ is the backtracking step that meets the index of all exact matches in the optimal alignment.

The first step in the backtracking process is to find the last exact match in the optimal alignment M_{end} which is the exact match with the highest score such that $S_{end} = S$. All other exact matches of the optimal alignment are obtained by repeating the backtracking step. The backtracking process finishes when $i = W_i$ which indicates M_i can be considered as the first exact match in the alignment.

The complexity of the above algorithm is $O(3n)$ for sorting, $O(n(n-1))$ for dynamic programming, and $O(v)$ for backtracking where v is the number of exact matches in the optimal alignment which is much smaller than n. As a consequence, the total complexity is $O(n^2)$. The above algorithm is capable of returning all optimal alignments as well as the n^{th} best alignments by considering S as the n^{th} largest element of S_i. Details are discussed in *Supplementary Data* Section 10.4.

4.2 Alignment using maximal exact matches

Note that n can be reduced by processing only MEMs. If M_a and M_b are consecutive exact matches in an optimal alignment, there are MEMs M_{xa} and M_{xb} that include M_a and M_b as a subsequences, respectively. However, M_{xa} and M_{xb} might overlap together which is not supported by the dynamic programming algorithm above. Figure 4.2 is an example which represents the overlap between two consecutive maximal exact matches.

In order to deal with overlaps, the maximum overlapping length between M_j and M_i in T and Q (MO_i^j) is computed using Equation 4.2a. Since there could be no mismatches when an overlap exists $\hat{N}_x^{j,i}$ is replaced with $N_x^{j,i}$ computed using Equation 4.2b. Subsequently, $\hat{P}_i^j$ is replaced with P_i^j computed using Equation 4.2e. Since the score for an overlapping region should not be added twice, the length of a M_i excluding its overlap with M_j (L_i^j) is computed using Equation 4.2c. Also, $\hat{F}_i^j$ is replaced with F_i^j computed using Equation 4.2d to allow for overlaps. Finally $\hat{S}_i^j$, $\hat{S}_i$ and $\hat{S}$ are replaced with S_i^j, S_i and S computed using Equation 4.2f, 4.2g and 4.2h, respectively, to handle overlapping MEMs correctly.

When backtracking, $MO_i^{W_i}$ is computed, and added to BT_i and BQ_i to exclude overlapping regions if any are present.

$$MO_i^j = \max(EQ_j - BQ_i, ET_j - BT_i) + 1 \tag{4.2a}$$

$$N_x^{j,i} = \begin{cases} \hat{N}_x^{j,i} & MO_i^j = 0 \\ o & \text{otherwise} \end{cases} \tag{4.2b}$$

$$L_i^j = L_i - MO_i^j \tag{4.2c}$$

$$F_i^j = \begin{cases} true & EQ_j < EQ_i \wedge ET_j < ET_i \\ flase & \text{otherwise} \end{cases} \tag{4.2d}$$

$$P_i^j = (N_x^{j,i} \times P_x) + (N_g^{j,i} \times P_g) + \begin{cases} P_o & N_g^{j,i} \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.2e}$$

$$S_i^j = \begin{cases} S_j + (L_i^j \times R_m) - P_i^j & F_i^j = true \\ 0 & \text{otherwise} \end{cases} \tag{4.2f}$$

$$S_i = \max \begin{cases} \max\limits_{1 \leq j \leq i-1} S_i^j \\ (L_i \times R_m) \end{cases} \tag{4.2g}$$

$$S = \max_{1 \leq i \leq n} S_i \tag{4.2h}$$

5 Extraction of maximal exact matches

The list of all possible MEMs between two sequences is the input to the algorithm described in Section 4.2. However, the complexity of the extraction process may exceed the complexity of the alignment process if implemented by brute force. An example brute force extraction algorithm with complexity of $O(n^3)$ is described in *Supplementary Data* Section 10.5. In order to make the MEM-Align feasible, we have introduced a fast, bit-vector extraction procedure.

In our proposed MEM extraction method, the first step is to represent sequences with a bit-vector. For nucleotide sequences A, C, T, and G are encoded into 00, 01, 10, and 11. This encoding is chosen to take advantages of a fast, parallel and bitwise conversion process that transform DNA strings to a bit-vector. The conversion process is described in *Supplementary Data* Section 10.6. Figure 5.1 illustrates an example sequence pair, along with their corresponding bit-vector representation.

For each exact match M_i, offset OFS_i is computed using Equation 5.1. Given two sequences in bit-vector format, the bitwise procedure described in Algorithm 1 computes a bit-vector E in which the beginning and ending positions of all MEMs with offset zero are identified by a set bit (a bit with value of one). This procedure consists of three phases: Φ_1, Φ_2, and Φ_3. In Φ_1, an XOR operation compares sequences where matches result in 00 and mismatches result in 01, 10, and 11. Phase Φ_2 transforms all mismatches into 11. Finally, in Φ_3 a shift-and-XOR operation marks the edges of MEMs. Figure 5.1 is an example that shows the value of E at different phases.

In Figure 5.1, the i^{th} symbol of Q is compared with the i^{th} symbol of T resulting in only MEMs with offset zero being extracted. To obtain MEMs with offset sh the T bit-vector must be shifted sh symbols to the left, and Algorithm 1 must be repeated. Shifting T to the right results in extraction of MEMs with offset $-sh$. To extract all possible MEMs, T should be shifted to the left and right up to $len - 1$, one symbol at the time, where len is the length

Input: T, Q input sequences as bit-vectors
Output: E a bit-vector that marks MEMs

```
// $\mathbf{\Phi_1}$: Compare T and Q
$E \leftarrow T \oplus Q$;
// $\mathbf{\Phi_2}$: Transform mismatches into "11"
$E \leftarrow E \vee (E \gg 1)$;
$E \leftarrow E \vee ((E \wedge 0101...0101) \ll 1)$;
// $\mathbf{\Phi_3}$: Mark up MEM's edge
$E \Leftarrow E \oplus (E \gg 1)$;
```

Algorithm 1: Bit-wise MEM extraction

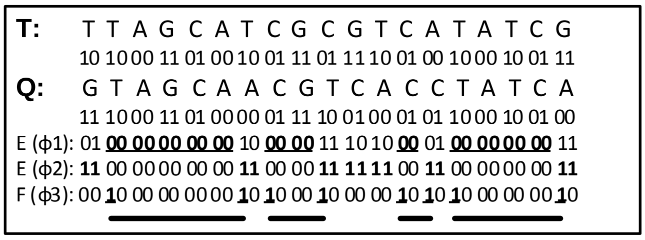

T: T T A G C A T C G C G T C A T A T C G
10 10 00 11 01 00 10 01 11 01 11 10 01 00 10 00 10 01 11
Q: G T A G C A A C G T C A C C T A T C A
11 10 00 11 01 00 00 01 11 10 01 00 01 01 10 00 10 01 00
E (ϕ1): 01 00 00 00 00 00 10 00 00 11 10 10 00 01 00 00 00 00 11
E (ϕ2): 11 00 00 00 00 00 11 00 00 11 11 11 00 11 00 00 00 00 11
F (ϕ3): 00 10 00 00 00 00 10 10 00 10 00 00 10 10 10 00 00 00 10

Figure 5.1: Representation of sequences with bit-vector. Identifying Edges of MEMs

$$OFS_i = BT_i - BQ_i = ET_i - EQ_i \tag{5.1}$$

of sequences (assuming both sequences are of the same length). When shifting T, the algorithm must only consider the overlapping part of the sequences using a mask bit-vector. In order to extract information from E and list the MEMs in triple integer format, the index of all set bits should be obtained. Details of the extraction process are explained in *Supplementary Data* Section 10.7.

6 Optimisation

The main body of the MEM-Align is a dynamic programming algorithm that is quadratic on the total number of extracted MEMs. Note that number of MEMs is much larger than the length of the sequences (see Table 10.1). Since the Smith-Waterman is a dynamic programming algorithm quadratic in the length of sequence, there is no reason to prefer MEM-Align over the Smith-Waterman algorithm. However, the algorithm allows a series of optimisations listed below, which significantly minimise the execution time of MEM-Align.

6.1 Reduced S_i^j computation

In order to compute S_i, S_i^j is computed for all $1 \leq j \leq (i-1)$ in the main body of MEM-Align, which results in an algorithm of complexity $O(n^2)$. However, there is a subset Ω_i of the set $\{M_1 \dots M_{i-1}\}$ which satisfies Equation 6.1. As a result, computing S_i^j for all j such that $M_j \in \Omega_i$ would be enough to find the maximum S_i^j. To better understand how Ω_i is defined, a diagram in Figure 6.1 represents a set of MEMs as lines where MEMs in a row have the same offset. The placement of lines in each row represents the placement of the related MEMs in Q. In Figure 6.1, bold lines represent Ω_i. These MEMs that are not involved in the computation of S_i are drawn in grey. Ω_i has at most one member in each row. $M_{\omega(i,ofs)}$ is a member of Ω_i if it is the closest to M_i amongst other MEMs in the same row. Also, $M_{\omega(i,ofs)}$ should not be fully overlapped by M_i. Formal definition, proof and implementation details are provided in Section 7.1.

6.2 Skipping distant MEM

When $N_x^{j,i}$ has a large value, P_i^j tends to be large which leads to a small value for S_i^j. Such small S_i^j is less likely to maximise S_i. As a consequence, the algorithm skips the computation of S_i^j where $N_x^{j,i}$ is greater than a threshold TD. With a suitable value selected for TD, there will be rare cases where distant MEM optimisation results in a sub-optimal alignment. $N_g^{j,i}$ is not restricted in distant MEM optimisation as this might prevent identifying alignments with large gaps.

6.3 Gap limited alignment

Gap limited alignment (also known as banded alignment) has been implemented in most alignment functions of the SeqAN [22] package as well as in BWA-MEM [23]. In fact, for a realistic dataset, most alignments are not likely to contain lengthy gaps except where Copy Number Variations (CNVs) occur. Note that CNVs are treated and identified differently from indels (insertion and deletions) which are expected to be identified during the alignment. Gap limited optimisation is applied to traditional dynamic programming alignment by avoiding computing entries in the dynamic programming table where the distance to the diagonal of the table is higher than a specific threshold gl. A similar optimisation is applied to MEM-Align by restricting the MEM extraction process so that T is only shifted up to gl to the right and the left. Limiting shifts not only reduces the number of MEMs to be processed but also speeds up

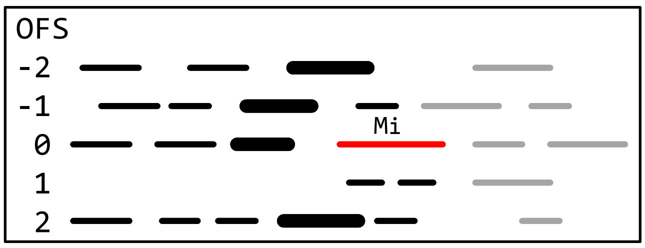


Figure 6.1: Representation of MEMs with lines. Each row contains MEMs with the same offset in the same order they appear in Q. Black lines represent M_j where $j < i$. Set Ω_i is shown with bold black line.

$$\max_{M_j \in \Omega_i} S_i^j = \max_{1 \le j \le i-1} S_i^j \tag{6.1}$$

the extraction process itself. How gap limited optimisation affects the output alignment is described in Section 7.2.

6.4 Short MEM removal

While the number of MEMs which exist between a pair of sequences might be quite high, the number of exact matches that form the optimal alignment is, in contrast, much smaller. This difference indicates that the majority of extracted MEMs are not a part of the optimal alignment. The main reason for such an excess of MEMs is the high number of short MEMs that randomly exist between a pair of sequences. Another bitwise operation is proposed in Algorithm 2 to be incorporated into Algorithm 1 to mask MEMs shorter than threshold sl prior to extraction. This modification to the MEM extraction process is discussed in Section 7.3.

Since there are cases in which short MEMs are part of an optimal alignment, this optimisation introduces a considerable inaccuracy in the alignment process. However, this adverse effect is mostly compensated with a modification to the original algorithm which is explained in Section 7.4.

Another challenge is to find a proper value for sl. The most logical way to do this is to find the probability that a random sequence of length shorter than sl is found in a random sequence of length $sl + (2 \times gl)$. However computing this probability is another difficult problem to solve. On the other hand, the value of sl is expected to be small, i.e. less than ten. As a consequence, the value of sl is better identified empirically using a try and test method.

6.5 Hybrid alignment

Even considering the optimisations introduced in Section 6.3 and 6.4, the number of MEMs is still too high when highly repetitive sequences are given as input. Since such sequences are less probable, the Smith-Waterman algorithm is used to align the input if the number of extracted MEMs exceeds a threshold TM. Note that Smith-Waterman processes all sequences in a static time independent of their content.

7 Methods

This section provides the formal definition of the algorithm, proof and implementation details.

7.1 Reduced S_i^j computation

In order to prove Equation 6.1 which is the basis of the optimisation introduced in Section 6.1, the following sets are defined first:

- H^i_{ofs}: a set of M_j such that $OFS_j = ofs$ and $EQ_j < BQ_{\omega(i,ofs)}$. In other words, a set of MEMs in the same row as $M_{\omega(i_ofs)}$ and appears before $M_{\omega(i,ofs)}$.

- H^i_*: a set of M_j such that $j < i$ and $BQ_j > BQ_i$. In other words, a set of MEMs which are fully overlapped by M_i.

Note that regardless of ofs, there is no common element between Ω_i, H^i_{ofs} and H^i_* whereas their union is equal to $\{M_j : j < i\}$. For example, in Figure 6.1 all the black lines after the bold lines (Ω_i) are members of H^i_* and all the lines which appear before the bold lines in the row ofs are members of H^i_{ofs}.

The next step is to prove Inequality 7.1a and Inequality 7.1b. Inequality 7.1a states that for all M_j in H^i_{ofs} there is a member $M_{\omega(i,ofs)}$ of Ω_i such that $S_i^{\omega(i,ofs)}$ is larger or equal to S_i^j; thus, it is possible to avoid computation of S_i^j if M_j is a member of H^i_{ofs}. Inequality 7.1b states that for all M_j in H^i_* there exists M_k such that S_i^k is larger or equal to S_i^j; thus, it is possible to avoid computation of S_i^j if M_j is a member of H^i_*. As a result, to prove Equation 6.1 it is sufficient to prove Inequality 7.1a and Inequality 7.1b. In the proof, F_i^j is assumed to be true for all j and i; otherwise computation of S_i^j is avoided as a consequence of its definition.

$$\forall M_j \in H^i_{ofs}(S_i^{\omega(i,ofs)} \geq S_i^j) \tag{7.1a}$$
$$\forall M_j \in H^i_* \, \exists M_k(S_i^k \geq S_i^j \land S_j = S_j^k) \tag{7.1b}$$

For clarity let ω be equal to $\omega(i, ofs)$. By Equation 4.2f, Equation 7.2a and 7.2b hold; thus proving Inequality 7.1a reduces to proving Inequality 7.2c. Since by Equation 4.2g, S_ω is larger or equal to S^j_ω, it suffices to prove Inequality 7.2d. Using Equation 7.2e, Inequality 7.2d reduces to Inequality 7.2f.

$$S_i^\omega = S_\omega + (L_i^\omega \times R_m) - P_i^\omega \tag{7.2a}$$
$$S_i^j = S_j + (L_i^j \times R_m) - P_i^j \tag{7.2b}$$
$$S_\omega + (L_i^\omega \times R_m) - P_i^\omega \geq S_j + (L_i^j \times R_m) - P_i^j \tag{7.2c}$$
$$S_\omega^j + (L_i^\omega \times R_m) - P_i^\omega \geq S_j + (L_i^j \times R_m) - P_i^j \tag{7.2d}$$
$$S_\omega^j = S_j + (L_\omega^j \times R_m) - P_\omega^j \tag{7.2e}$$
$$(L_\omega^j + L_i^\omega - L_i^j) \times R_m \geq P_\omega^j + P_i^\omega - P_i^j \tag{7.2f}$$

In a similar fashion Inequality 7.1b is reduced to Inequality 7.3f using Equation 7.3a and 7.3b and Inequality 7.3c and 7.3d as well as Equation 7.3e, with the difference that in Inequality 7.3c S_j is replaced with S_j^k to form Inequality 7.3d. Note that S_j is equal to S_j^k due to the assumption in Inequality 7.1b and Equation 4.2g.

$$S_i^k = S_k + (L_i^k \times R_m) - P_i^k \tag{7.3a}$$
$$S_i^j = S_j + (L_i^j \times R_m) - P_i^j \tag{7.3b}$$
$$S_k + (L_i^k \times R_m) - P_i^k \geq S_j + (L_i^j \times R_m) - P_i^j \tag{7.3c}$$
$$S_k + (L_i^k \times R_m) - P_i^k \geq S_j^k + (L_i^j \times R_m) - P_i^j \tag{7.3d}$$
$$S_j^k = S_k + (L_j^k \times R_m) - P_j^k \tag{7.3e}$$
$$P_j^k + P_i^j - P_i^k \geq (L_j^k + L_i^j - L_i^k) \times R_m \tag{7.3f}$$

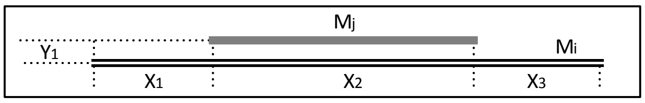


Figure 7.1: Categorization of MEMs based on their offset. The set Ω_i is shown in bold.

$$S_i^j \geq L_i \times R_m \tag{7.5a}$$
$$S_j + (L_i^j \times R_m) - P_i^j \geq L_i \times R_m \tag{7.5b}$$
$$(L_j \times R_m) + (L_i^j \times R_m) - P_i^j \geq L_i \times R_m \tag{7.5c}$$
$$((L_j + L_i^j) \times R_m) - P_i^j \geq L_i \times R_m \tag{7.5d}$$
$$((X_1 + X_2) \times R_m) - (Y1 \times P_e) - P_o \geq (X_1 + X_2 + X_3) \times R_m \tag{7.5e}$$

Now that we reduce Inequality 7.1a and 7.1b to Inequality 7.2f and 7.3f, respectively, it is time to verify Inequality 7.2f and 7.3f for all possible arrangements of MEMs. These verifications are moved to *Supplementary Data* Section 10.8 due to lack of space. These verifications are based on the line representation of MEMs (i.e. Figure 6.1) in which the horizontal and vertical spaces between two MEMs are the numbers of mismatches and gaps between them, respectively. Note that the horizontal space is equal to $BQ_\beta - EQ_\alpha - 1$ which is the definition of $N_x^{\alpha,\beta}$. The vertical space is the difference between MEMs offsets, $|OFS_\beta - OFS_\alpha|$ which is proven to be equal to $N_g^{\alpha,\beta}$ in Equation 7.4 using Equation 5.1.

$$\begin{aligned} |OFS_\beta - OFS_\alpha| &= |(BT_\beta - BQ_\beta) - (ET_\alpha - EQ_\alpha)| \\ &= |(BT_\beta - ET_\alpha) - (BQ_\beta - EQ_\alpha)| \\ &= |LT_\alpha^\beta - LQ_\alpha^\beta| \\ &= N_g^{\alpha,\beta} \end{aligned} \tag{7.4}$$

There is one exception which is not considered in Inequality 7.1b, namely the case that there is no M_k such that $S_j = S_j^k$. This case occurs when S_j is obtained by taking M_j as the first MEM in the alignment which means S_j is equal to (or is maximised by) $L_j \times R_m$. On the other hand, considering M_i as the first MEM in the alignment results in $S_i = L_i \times R_m$. Based on Equation 4.2g, computation of S_i^j is unavoidable if S_i^j is the largest value for S_i; subsequently, larger than $L_i \times R_m$ as well. This condition is shown in Inequality 7.5a which is reduced to Inequality 7.5b using Equation 4.2f and then to Inequality 7.5c using the exception condition where $S_j = L_j \times R_m$. Inequality 7.5c is then reduced in Inequality 7.5d.

The computation of S_i^j is avoidable in this exceptional case, if Inequality 7.5d does not hold. By substituting values from Figure 7.1 which is the only possible arrangement of MEMs in this exceptional case, Inequality 7.5d yields 7.5e which can not hold if $R_m > 0$, $P_x > 0$, $P_o > 0$ and $P_g > 0$

7.2 Gap limited alignment

Gap limited optimisation does not necessarily limit the length of the gap to gl. To understand how this optimisation limits the output alignment, INS_i and DEL_i are defined as the total length of insertions and deletions from the beginning of the alignment up to the i^{th} symbol in the alignment. Then G_i is defined as $INS_i - DEL_i$. When gap limited alignment is applied, in the output alignment the value of G_i is always bounded by $-gl$ and gl.

If the optimal alignment does not satisfy the condition above, it cannot be found using gap limited optimisation. However, in real datasets, lengthy gaps are rare and choosing a proper value for gl should result in a negligible probability of an optimal alignment being missed.

7.3 Masking short MEMs from the edge bit-vector

In order to mask short MEMs during the MEM extraction process described in Section 5, our proposed solution is to mask short MEMs in the E bit-vector of Algorithm 1 before phase Φ_3 where the edges of MEMs are marked. The procedure shown in Algorithm 2 takes place between phases Φ_2 and Φ_3 of Algorithm 1 and replaces MEMs shorter than sl with mismatches. As a consequence, MEMs shorter than sl are not identified in phase Φ_3 of Algorithm 1. The short MEM removal procedure in Algorithm 2 consist of two sub-processes: ϕ_1 and ϕ_2. In ϕ_1 a bit-vector F is formed in which each match (00) indicates that the next $sl-1$ symbols on the right were matched. In ϕ_2 each match in the F bit-vector is extended to the right by $sl-1$ symbols to form the modified E bit-vector in which MEMs shorter than sl are masked.

Although masking short MEM comes with additional processing, its overall effect on execution time is positive because there would be less MEMs in each bit-vector to list and also less MEMs in total to be subsequently processed.

7.4 Short MEM Removal

There are situations where short MEMs can appear in the optimal alignment. If these short MEMs are removed by optimisation in Section 6.4, the algorithm in Section 4.2 can no longer identify optimal alignments. We present modifications to the original algorithm to deal with some of these situations.

The simplest case where a short MEM occurs in an optimal alignment is when two edits are close enough to form a MEM shorter than sl. In a more

```
Input: $E$ edge bit-vector
Input: $sl$ short MEM length
Output: $E$ edge bit-vector, masked short MEM
// $\phi_1$: Forming F
$F \leftarrow E$;
for $i \in \{1, \ldots, sl-1\}$ do
    $F \leftarrow F \vee (E \ll (2 \times i))$;
end
// $\phi_2$: Forming modified E
$E \leftarrow F$;
for $i \in \{1, \ldots, sl-1\}$ do
    $E \leftarrow E \wedge (F \gg (2 \times i))$;
end
```

Algorithm 2: Short MEM Removal

complex case, concentration of more than two edits in a narrow region (REG) results in consecutive short MEMs in the optimal alignment resulting in all of them being eliminated. Figure 7.2 represents four general cases where short MEMs appear in the optimal alignment. In all cases, M_j and M_i as well as grey (eliminated) short MEMs are part of optimal alignment. In case 1, only mismatches exist, while in case 2, one gap (here gap refers to a continuous gap of any length) on one side of the REG also exists. In case 3, there is one gap but in the middle of REG. Case 4 represents the situation in which two or more gaps exist in the REG.

Since $N_x^{j,i}$ and $N_g^{j,i}$ can no longer be computed correctly using Equation 4.2b and 2.2f respectively, computing P_i^j using Equation 4.2e is not possible. Considering M_j and M_i consecutive long MEMs in an optimal alignment, the distance between them could not be assumed as all mismatches and gaps as there could have been short MEMs between them that were removed. In fact, depending on the value of sl the overall score for region REG could be positive but end up being represented with a negative P_i^j.

In order to deal with all of the above cases, P_i^j should be computed using a global optimal alignment that forces all symbols in REG to be aligned to the corresponding region in T. However, this solution is time-consuming as it should be executed each time P_i^j is computed. As a result, we propose a faster method to retrieve the correct P_i^j which only supports the first two cases of Figure 7.2.

Note that the rate of gaps in the alignment is much smaller than the rate of substitutions; thus the probability of having two gaps near each other is extremely small. Also, as the gap open penalty is usually high, the optimal

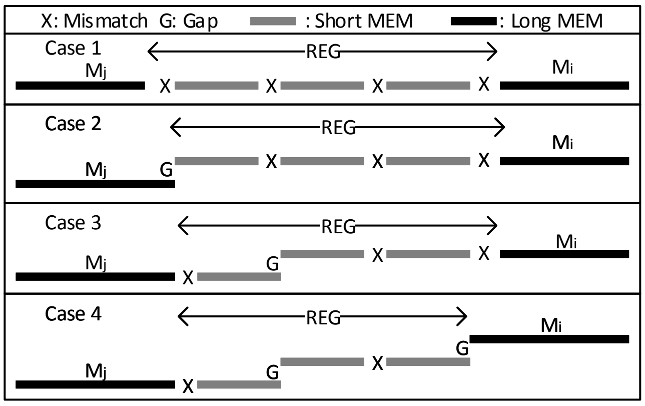


Figure 7.2: Generalized cases where two or more edits result in short MEMs in an optimal alignment. There could be multiple consecutive short MEMs that result in elimination of region REG

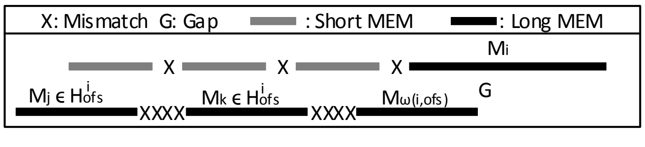


Figure 7.3: The need for extension of Ω_i when short MEMs are removed.

alignment tends to put gaps together, rather than leaving multiple separate gaps in a narrow region. Case 3 in Figure 7.2 has a similar probability as case 2 but we neglect it for the sake of performance of our aligner. Experimental results in Section 8 shows that our proposed method delivers acceptable accuracy on a realistic dataset.

Since in the first two cases at most one gap is assumed, $N_g^{j,i}$ can be computed correctly as in Equation 2.2f. To retrieve the number of mismatches $\bar{N}_x^{j,i}$ and matches $N_m^{j,i}$ between M_j and M_i, our proposed method is to look back into subsequences of T and Q (TS_i^j and QS_i^j respectively) which are bounded by M_j and M_i. If there is no gap TS_i^j and QS_i^j are of the same size ($LT_i^j = LQ_i^j$) and there is only one way to align them; thus $\bar{N}_x^{j,i}$ and $N_m^{j,i}$ are counted by comparing symbols in TS_i^j and TQ_i^j one by one sequentially. If there is a gap, assuming that the gap is attached to either M_j or M_i, there are only two alignments, that is, aligning TS_i^j and TQ_i^j to the left and right of each other. $\bar{N}_x^{j,i}$ and $N_m^{j,i}$ are counted in the overlapping region of TS_i^j and TQ_i^j. The alignment that results in the lower $\bar{N}_x^{j,i}$ is chosen.

$\bar{P}_i^j$ is computed using Equation 7.6 based on the value of $N_g^{j,i}$, $\bar{N}_x^{j,i}$ and $N_m^{j,i}$, $\bar{P}_i^j$. $\bar{P}_i^j$ should then replace P_i^j in Equation 4.2f. To save space and for simplicity we do not redefine subsequent equations with annotated names.

$$\bar{P}_i^j = (\bar{N}_x^{j,i} \times P_x) + (N_g^{j,i} \times P_g) + \begin{cases} 1 & N_g^{j,i} \neq 0 \\ 0 & \text{otherwise} \end{cases} - (N_m^{j,i} \times R_m) \tag{7.6}$$

Comparing TS_i^j and TQ_i^j in a sequential manner for all computed P_i^j is yet another time-consuming process; thus we propose an optimisation to avoid computing P_i^j in some cases. In the proposed optimisation we compute minimum possible P_i^j and check if the resulting S_i^j is higher than the current maximum computed S_i. If this condition is met, the actual P_i^j is computed and S_i^j is compared to the current maximum S_i.

In order to estimate the minimum possible P_i^j, we assume REG is composed mainly of groups of contiguous $sl - 1$ matches which are separated by individual mismatches. On both sides of REG there should be a gap or a mismatch to separate it from the rest of the alignment. In each sl group of symbol there should be at lest one mismatch. Based on the number of remaining symbols (computed using equation 7.7a and the existence of the gap on one side of REG one or two additional mismatches might be added to the end. the maximum possible $N_m^{j,i}$ ($maxN_m^{j,i}$) and subsequently minimum possible $\bar{N}_x^{j,i}$ ($min\bar{N}_x^{j,i}$) in REG are then computed using Equation 7.7b and 7.7c respectively, based on the length of the REG which is given by $L_{reg} = min(LT_i^j, LQ_i^j)$ and the parameter sl.

$$MOD = L_{reg} \mod sl \tag{7.7a}$$

$$min\bar{N}_x^{j,i} = \left\lfloor \frac{L_{reg}}{sl} \right\rfloor + \begin{cases} 1 & MOD = 0 \wedge N_g^{j,i} = 0 \\ 1 & MOD = 1 \wedge N_g^{j,i} = 0 \\ 2 & MOD \geq 1 \wedge N_g^{j,i} = 0 \\ 0 & MOD = 0 \wedge N_g^{j,i} \geq 0 \\ 1 & MOD = 1 \wedge N_g^{j,i} \geq 0 \\ 1 & MOD \geq 1 \wedge N_g^{j,i} \geq 0 \end{cases} \tag{7.7b}$$

$$maxN_m^{j,i} = L_{reg} - min\bar{N}_x^{j,i}; \tag{7.7c}$$

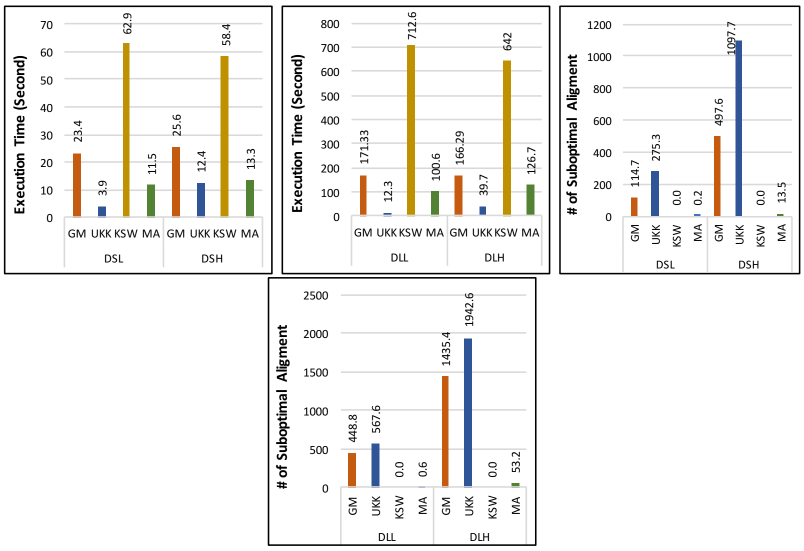


Figure 7.4: Execution time and number of non-optimal alignments obtained using GM, UKK, KSW and MA (MEM-Align) for datasets listed in Table 8.1

Short MEMs might appear before or after the first and the last MEM of the optimal alignment. This issue is treated similarly to short MEMs between two long MEMs as discussed above. After the optimal alignment has been found, the subsequences of T and Q that appears on the left of the first MEM of the optimal alignment M_{first} are aligned to the right of each other. Then, the numbers of matches and mismatches are counted from right to left for the whole overlapping region. The match score and mismatch penalty are progressively added to and subtracted from S_{first} to see at which point it has been maximised. Finally the beginning of M_{first} in T and Q is extended left-ward to the point that maximum S_{first} is achieved. A similar procedure is applied to the last MEM of the optimal alignment in the reverse direction.

The list Ω_i should be reconsidered when short MEMs are removed. The example in Figure 7.3 represents a case where $S_i^{\omega(i,ofs)}$ does not maximise S_i. In this example, the area before M_i in the same row is full of matches with a minimal number of mismatches which results in elimination of the whole region. In this example, the true score for this region is obtained by computing P_i^j. However, as M_j is not a member of Ω_i, S_i^j and subsequently P_i^j are not computed when the optimisation proposed in Section 6.1 is applied. Since such situation is rare, and the optimisation introduced in Section 6.1 has a good effect on final execution time, we only slightly extend ω_i. In fact for all $M_\omega(i, ofs) \in \Omega_i$, if $M_\omega(i, ofs)$ overlaps M_i, the immediately previous MEM with offset ofs (M_k in the example) is added to Ω_i. Note that in our example in Figure 7.3, this extension does not fix the problem. However, this extension of Ω_i is enough to correctly deal with most real case alignments.

8 Experimental results

In order to evaluate MEM-Align, four synthetic datasets, shown in Table 8.1, were prepared by random selection from the reference human genome followed by simulated variation. Each dataset contained two million sequence pairs, with sequence length and variation rate varying between datasets. These multiple datasets allowed estimating the impact of sequence length and sequence divergence levels on the speed and accuracy of the various algorithms. More details about input preparation can be found in *Supplementary Data* Section 10.1. All the tests were run on a Linux (version 3.13.0-58-generic) machine with Intel X7560 processors, in single thread mode. The Linux *perf* tool was used for measuring the execution time of each program.

Table 8.1: Datasets

Dataset	Sequence Length	Variation Rate		
		SNP	Indel	Indel Expansion
DSL	125	0.01	0.001	0.05
DLL	500	0.01	0.001	0.05
DSH	125	0.05	0.005	0.1
DLH	500	0.05	0.005	0.1

MEM-Align is a combination of several processes including string to bit-vector conversion, MEM extraction and sorting, as well as MEM alignment. Each of these processes is affected by parameters such as the gap limit threshold gl, the short MEM removal length sl, the distant MEM threshold TD and the maximum allowed MEM threshold TM. Considering all possible configurations for these MEM-Align parameters and listing all the results is not practical. However, *Supplementary Data* Section 10.10, we present a collection of configuration that demonstrate the performance of most individual algorithmic features of MEM-Align.

We compared MEM-Align with three other alignment algorithms: vectorized Smith-Waterman (KSW), GeneMyer (GM) and Ukkonen (UKK). To evaluate the accuracy of each non-optimal method, their resulting alignment scores were compared with the optimal alignment scores produced by KSW. The number of non-optimally aligned sequence pairs was used as a metric to judge the accuracy of the corresponding algorithm. Figure 7.4 summarises the execution time of each method as well as the number of non-optimal alignments produced by each method. Figure 7.4. For MEM-Align, parameter values (gl, sl, TM, TD) are set to (5,4,100,20) for DSL and DSH, and are set to (10,4,300,20) for DLL and DLH. These values were selected in order to balance speed and accuracy. This configuration of MEM-Align resulted in an approximately 5-7 times speedup over vectorized Smith-Waterman, with only a small decrease in alignment quality. More details regarding implementation of these algorithms and our usage are given in *Supplementary Data* Section 10.9.

9 Discussion

The MEM extraction process in MEM-Align has a lot in common with the Shifted Hamming Distance (SHD) introduced in [24]. In [25], it was shown that SHD is suitable for acceleration using custom hardware, and the MEM-Align extraction process is similarly well-suited for hardware acceleration due to its similarity to SHD. Furthermore, the method we propose for masking short MEM seems to be more flexible and more efficient than the method used in SHD. In the case of protein sequences the normal shift-and-insert process should be employed to transform protein sequences into bit-vectors. Since, in protein sequences, the mismatch penalty varies based on the symbols, those methods which we used for DNA sequences to compute the score for the region between two MEMs (P_i^j) are not applicable. As a consequence, a form of global alignment is required to align the regions between MEMs.

References

[1] Sean R Eddy. "What is dynamic programming?" In: *Nature Biotechnology* 22.7 (July 2004), pp. 909–910.

[2] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". In: *Journal of molecular biology* 48.3 (1970), pp. 443–453.

[3] T.F. Smith and M.S. Waterman. "Identification of common molecular subsequences". In: *Journal of Molecular Biology* 147.1 (Mar. 1981), pp. 195–197.

[4] Stephen F Altschul et al. "Basic local alignment search tool". In: *Journal of molecular biology* 215.3 (1990), pp. 403–410.

[5] Heng Li and Richard Durbin. "Fast and accurate long-read alignment with Burrows-Wheeler transform." In: *Bioinformatics (Oxford, England)* 26.5 (Mar. 2010), pp. 589–95.

[6] Stefan Kurtz et al. "Versatile and open software for comparing large genomes." In: *Genome biology* 5.2 (Jan. 2004), R12.

[7] Ben Langmead et al. "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome." In: *Genome biology* 10.3 (Jan. 2009), R25.

[8] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2." In: *Nature methods* 9.4 (Apr. 2012), pp. 357–9.

[9] Mark A DePristo et al. "A framework for variation discovery and genotyping using next-generation DNA sequencing data." In: *Nature genetics* 43.5 (May 2011), pp. 491–8.

[10] Yongchao Liu and Bertil Schmidt. "Long read alignment based on maximal exact match seeds." In: *Bioinformatics (Oxford, England)* 28.18 (Sept. 2012), pp. i318–i324.

[11] Michael Farrar. "Striped Smith-Waterman speeds database searches six times over other SIMD implementations". In: *Bioinformatics* 23.2 (Jan. 2007), pp. 156–161.

[12] Mengyao Zhao and et al. Lee. "SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications". In: *PLoS ONE* 8.12 (Dec. 2013). Ed. by Leonardo Mariño-Ramírez, e82138.

[13] Jeff Daily and et al. Henikoff. "Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments". In: *BMC Bioinformatics* 17.1 (Dec. 2016), p. 81.

[14] Adam Szalkowski et al. "SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2." In: *BMC research notes* 1 (2008), p. 107.

[15] Y Liu, A Wirawan, and B Schmidt. "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions". In: *BMC Bioinformatics* 14 (2013), p. 117.

[16] Brandon Harris et al. "A Banded Smith-Waterman FPGA Accelerator for Mercury BLASTP". In: *2007 International Conference on Field Programmable Logic and Applications.* IEEE, Aug. 2007, pp. 765–769. ISBN: 978-1-4244-1059-0.

[17] Jeff Allred et al. "Smith-Waterman implementation on a FSB-FPGA module using the Intel Accelerator Abstraction Layer". In: *2009 IEEE International Symposium on Parallel & Distributed Processing.* IEEE, May 2009, pp. 1–4. ISBN: 978-1-4244-3751-1.

[18] Santiago Marco-Sola et al. "The GEM mapper: fast, accurate and versatile alignment by filtration." In: *Nature methods* 9.12 (Dec. 2012), pp. 1185–8.

[19] Matei Zaharia et al. "Faster and More Accurate Sequence Alignment with SNAP". In: *arXiv* (Nov. 2011).

[20] Gene Myers. "A fast bit-vector algorithm for approximate string matching based on dynamic programming". In: *Journal of the ACM* 46.3 (May 1999), pp. 395–415.

[21] Esko Ukkonen. "Algorithms for approximate string matching". In: *Information and Control* 64.1 (1985), pp. 100–118.

[22] Andreas Döring and et al. Weese. "SeqAn An efficient, generic C++ library for sequence analysis". In: *BMC Bioinformatics* 9.1 (2008), p. 11.

[23] Heng Li. "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM". In: (Mar. 2013), p. 3.

[24] Hongyi Xin et al. "Shifted Hamming distance: a fast and accurate SIMD-friendly filter to accelerate alignment verification in read mapping." In: *Bioinformatics (Oxford, England)* 31.10 (May 2015), pp. 1553–60.

[25] Mohammed Alser et al. "GateKeeper : Enabling Fast Pre-Alignment in DNA Short Read Mapping with a New Streaming Accelerator Architecture". In: (2016).

[26] Nils Homer. *Whole Genome Simulator for Next-Generation Sequencing.* 2011. URL: `https://github.com/nh13/DWGSIM`.

10 Supplementary Data

10.1 Input Preparation

In order to prepare input sequence pairs, we simulate 600 bp reads from human genome using DWGsim [26] with zero error rates. The generated FastQ file is then transformed into a Fasta file. This Fasta file contains target sequences. Using EditSim, edits are injected (at two different rates) to the target Fasta to create two query Fasta file containing query sequences. EditSim source code is bundled with MEM-Align. The target and query Fasta files are then trimmed to 125 bp and 500 bp to be processed in our evaluation program. The Linux script named as "PrepareData.sh" is included in MEM-Align package and elaborates details of input preparation process.

10.2 Sorting

There is a wide range of sorting algorithm for various applications. In the case of MEM-Align that needs the extracted MEMs to be sorted in order of EQ, the counting sort is chosen. Counting sort is a linear time sorting algorithm with complexity of $O(3n) = O(n)$ that is applicable only when the sorted parameter varies in a small range. Since the EQ varies between 1 and sequence length and the sequence length is a relatively small value, the counting sort strategy is well suited to this problem.

Counting sort consist of three steps: the number of MEMs ending at each position is computed in array A_{cnt} (Δ_1); the cumulative number of MEMs ending before each position is computed in array A_{cum} (Δ_2); for each MEM its index in the sorted list (SLI) is identified and the MEM is copied to its place in the sorted list SL (Δ_3). Algorithm 3 clarifies the sorting process.

10.3 Global Alignment

The dynamic programming algorithm that processes MEMs is described in Section 8. Here we explain how this algorithm could be modified to produce a global alignment where all symbols in Q must be included in the alignment. This does not necessarily mean that the first and the last exact matches of the alignment must begin and end from the start of Q and to the end of Q. In fact there are optimal global alignments that consider leading and trailing symbols as mismatches or insertions (whatever leads to lower penalty). As a result, our global alignment should allow the alignment to begin and end with any MEMs similar to the proposed local alignment. However, when computing S_i, considering M_i as the first MEM in the alignment requires considering the penalty for all symbols which comes before M_i in Q. To apply this change S_i and S are redefined in Equation 10.1.

```
Input: $\{M_1 \dots M_n\}$ List of MEMs
Output: $SL$ Sorted List of MEMs
// $\mathbf{\Delta_1}$: Compute $\boldsymbol{A_{cnt}}$
$A_{cnt} \leftarrow [0, 0, 0, \dots, 0]$;
for $i \in \{1, \dots, n\}$ do
    $A_{cnt}[EQ_i] \leftarrow A_{cnt}[EQ_i] + 1$;
end
// $\mathbf{\Delta_2}$: Compute $\boldsymbol{A_{cum}}$
$A_{cum}[1] \leftarrow 0$;
for $i \in \{2, \dots, n\}$ do
    $A_{cum}[i] \leftarrow A_{cum}[i-1] + A_{cnt}[i-1]$;
end
// $\mathbf{\Delta_3}$: Sort MEMs
$A_{tmp} \leftarrow [0, 0, 0, \dots, 0]$;
for $i \in \{1, \dots, n\}$ do
    $SLI_i \leftarrow A_{cum}[EQ_i] + A_{tmp}[EQ_i]$;
    $SL[SLI_i] \leftarrow M_i$;
    $A_{tmp}[EQ_i] \leftarrow A_{tmp}[EQ_i] + 1$;
end
```

Algorithm 3: Sort MEM by EQ

$$GLP_i = \min \begin{cases} ((BQ_i) - 1) \times P_g + P_o \\ ((BQ_i) - 1) \times P_x \end{cases} \tag{10.1a}$$

$$S_i = \max \begin{cases} \max\limits_{1 \le j \le i-1} S_i^j \\ (L_i \times R_m) - GLP_i \end{cases} \tag{10.1b}$$

$$GTP_i = \min \begin{cases} (Len - EQ_i) \times P_g + P_o \\ (Len - EQ_i) \times P_x \end{cases} \tag{10.1c}$$

$$S = \max_{1 \le i \le n} (S_i - GTP_i) \tag{10.1d}$$

10.4 Backtracking

Another issue is how to obtain all n^{th} best alignments. In the Smith-Waterman algorithm this can be done by recursively backtracking in the table from all entries whose value is equal to the n^{th} maximum. While backtracking, the maximum values for each entry might be derived from multiple other entries with each of them resulting in a different alignment with the same score. In MEM-Align, for each MEM M_i, the algorithm stores all js for which $S_i^j = S_i$. Then for each MEM M_i where S_i is equivalent to the n^{th} maximum score the algorithm start backtracking from M_i. Multiple backtracking paths are processed when multiple js are stored for a MEM.

10.5 Brute force MEM extraction

Algorithm 4 represents a brute force method to extract all possible MEMs from a pair of sequences. In Algorithm 4, we consider each position in Q against each position in T to see if an exact match begins from that point. If $Q[i] \neq T[j]$ then no match is started; otherwise if $Q[i-1] = T[j-1]$ then the exact match is started from the previous position and is already extracted. If $Q[i] = T[j]$ and the previous symbols are not matched then it is the beginning of a maximal exact match. Thus we extend the exact match up to the end of T or Q, whatever comes first. The γ symbol at the end and beginning of T is to guarantee that a mismatch always appears before and after each MEM.

```
Input: $T, Q$ sequences of length $n$
Output: $List$ a list of MEMs
// Seal $T$ with $\gamma$ which is not a sequence alphabet
$T[0] \leftarrow \gamma$;
$T[n+1] \leftarrow \gamma$;
for $i \in \{1, \dots, n\}$ do
    for $j \in \{1, \dots, n+1\}$ do
        if $Q[i] \neq T[j] \vee Q[i-1] = T[j-1]$ then
            continue
        end
        $k \leftarrow i+1$;
        $l \leftarrow j+1$;
        while $k \leq n \wedge l \leq n+1$ do
            if $Q[k] \neq T[l]$ then
                $M.BQ \leftarrow i$;
                $M.BT \leftarrow j$;
                $M.L \leftarrow k - i$;
                $Insert(List, M)$;
                $break$;
            end
            else
                $k \leftarrow k+1$;
                $l \leftarrow l+1$;
            end
        end
    end
end
```

Algorithm 4: Brute force MEM extraction with complexity $O(n^3)$

10.6 DNA string to bit-vector conversion

Given a pair of sequences, the first processing step is to transform sequences into bit-vectors. A Bit-vector is stored as array of longest machine word, i.e. an array of 64-bit words each of which stores up to 32 symbols (assuming 2-bit per symbol for nucleotide sequences). For this transformation, a regular method is to shift-and-insert symbols into the data-word one by one. However, we propose a fast bit-vector method to convert nucleotide sequences into bit-vectors.

The third and second rightmost bits of the ASCII code for A, C, T, and G form four different combinations: 00, 01, 10, and 11 respectively. These values remain the same even if lower case letters are used. As a result, considering a 64-bit machine word W as an array of eight ASCII nucleotide symbols (each 8-bit long), the procedure represented in Algorithm 5 illustrates a method to transform all eight symbols into a 16-bit bit-vector located in the leftmost part of W. Finally, every four consecutive words are merged together to compress all 32 symbols into one 64-bit machine word.

Note that when copying an ASCII string into an array of 64-bit words, a little-endian machine such as Intel copies the left most symbol into the least significant byte of the first word in the array. As a consequence, Algorithm 5 reverses the sequence to store the leftmost symbol into the leftmost 2 bits of the output bit vector.

In order to demonstrate the performance of our proposed DNA-string to bit-vector conversion, all sequences from dataset DLL were converted to bit-vectors using our proposed method as well as a regular shift-and-insert Method. Both conversion methods are implemented in MEM-Align. The measured execution times were 28.4 and 8 seconds, respectively, which indicate an speed up of over 3.5 times in the conversion process.

```
Input: $IA$ eight ASCII code
Output: $W$ 2-bit encoded of $IA$
$W \leftarrow IA \wedge 0606060606060606(hex)$;
$W \leftarrow W \gg 1$;
$W \leftarrow W \vee (W \ll 10) \vee (W \ll 20) \vee (W \ll 30)$;
$W \leftarrow W \wedge FF000000FF000000(hex)$;
$W \leftarrow (W \gg 8) \vee (W \ll 32)$;
```

Algorithm 5: ASCII to Bit-Vector

10.7 Edge bit-vector to triple number representation of MEMs

The last step in the extraction process is to identify the beginning and the length of the MEMs in the bit-vector. Since phase Φ_3 of Algorithm 1 marks the beginning and ending of each MEM with a set bit, there is an even number of set bits in the bit-vector. Let SB_i be the position of the i^{th} set bit in bit-vector. The i^{th} MEM in the bit-vector M_i is marked by SB_{2i-1} and SB_{2i}. Parameters that describe M_i are then computed using Equation 10.2 where sh is the number of times T is shifted to the left by one prior to extraction (negative sh is considered for right shifts)

$$OFS_i = sh \tag{10.2a}$$

$$L_i = \frac{SB_{2i} - SB_{2i-1}}{2} \tag{10.2b}$$

$$BQ_i = \frac{SB_{2i-1}}{2} + 1 \tag{10.2c}$$

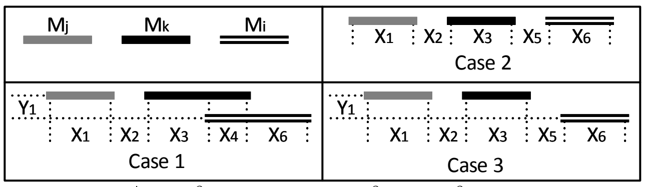
$$BT_i = BQ_i + OFS_i \tag{10.2d}$$

In order to compute SB for all set bits, one strategy is to shift out the bit-vector by two bits at a time and check if the left most bit is set. On average there should be few MEMs in a bit vector which means there are few set bits. As a consequence, shift-check loop is an inefficient solution. Since most modern processors have instructions to count the number of leading or trailing zeroes in a machine word, this operation is speeded up using these instructions. the Bit Scan Forward (BFS) and Bit Scan Reverse (BSR) instructions of Intel processors as well as the Leading Zero Count (LZCNT) instruction of ARM processors are just two examples.

10.8 Verification of Inequality 7.2f and 7.3f

The number of gaps and mismatches as well as the length of overlapping regions must be provided as they are used to compute P^{α}_{β} and L^{α}_{β} in Inequality 7.2f and Inequality 7.3f. We list all possible arrangements for M_i, M_j and M_k using a line representation. Finally, $N^{\alpha,\beta}_{x}$, $N^{\alpha,\beta}_{g}$ and $N^{\alpha,\beta}_{o}$ as well as L^{α}_{β} are extracted for each arrangement of MEMs and the corresponding inequality is evaluated.

Figure 10.1 illustrates all three possible arrangements of M_i, M_j and M_k (cases 1 to 3) that meet the condition of Equation 7.1a where M_j and M_k must be in the same row. When M_k overlaps M_i they cannot be in same row as M_i (case 1); otherwise M_k and M_j can be in same row as M_i (case 2) or in another row (case 3). Figure 10.1 represents the parameter used to evaluate Inequality 7.2f. Equation 7.2f is evaluated in all three cases and in all cases it results in $R_m \geq -P_x$ which is true for all $R_m > 0$ and $P_x > 0$. Note that during evaluation the value of X_u and Y_u is considered to be zero if not presented in the case.



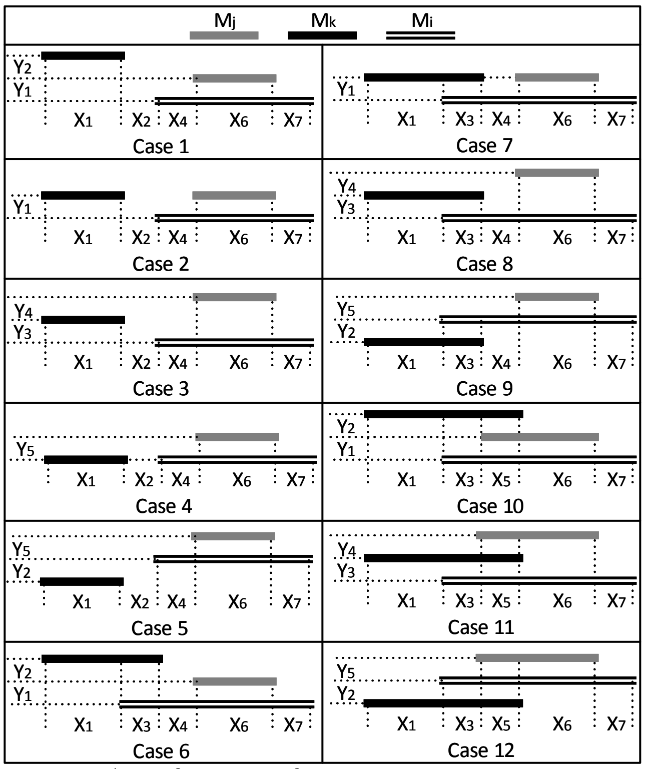
α	β	$N^{\alpha,\beta}_{x}$	$N^{\alpha,\beta}_{g}$	$N^{\alpha,\beta}_{o}$	L^{α}_{β}
j	k	X_2	0	0	$X_3 + X_4$
k	i	X_5	Y_1	1	X_6
j	i	$X_2 + X_3 + X_5$	Y_2	1	$X_4 + X_6$

Figure 10.1: Possible arrangements of MEMs for Equation 7.1a.

Figure 10.2 illustrates all twelve possible arrangements of M_i, M_j and M_k (cases 1 to 12) that meet the condition set in Equation 7.1b. M_j must be fully overlapped by M_i indicating that they should not be on the same row. For M_k, the three horizontal arrangements are: no overlaps of M_k with M_i and M_j; M_k overlaps only with M_i; and M_k overlaps both M_i and M_j.

When M_k does not overlap M_i and M_j there are five vertical arrangements for M_k: above M_j (case 1); same as M_j (case 2); between M_j and M_i (case 3); same as M_i (case 4); and below M_i (case 5). When M_k only overlaps M_i it cannot be in the same row as M_i which means that only four vertical arrangements are left (cases 6 to 9). When M_k overlaps both M_i and M_j it cannot be in the same row as M_i or M_j which means that only three vertical arrangements are left (cases 10 to 12).

Figure 10.2 represents the parameters used to evaluate Inequality 7.3f where $N_o^{\alpha,\beta}$ is 1 when $N_g^{\alpha,\beta} > 0$ and 0 otherwise. Equation 7.3f is evaluated in all twelve cases and the result for each case is presented in Figure 10.2. For all cases, Inequality 7.3f is evaluated as true if $R_m > 0$, $P_x > 0$, $P_g > 0$ and $P_o > 0$.



α	β	$N_x^{\alpha,\beta}$	$N_g^{\alpha,\beta}$	L_β^α
k	j	$X_2 + X_4$	$Y_2 + Y_4 + Y_5$	X_6
j	i	0	$Y_1 + Y_3 + Y_4 + Y_5$	X_7
k	i	X_2	$Y_1 + Y_2 + Y_3$	$X_4 + X_6 + X_7$

case 1:	$(X_4 \times P_x) + P_o \geq -(X_4 \times R_m)$
case 2:	$P_x \geq -R_m$
case 3:	$(X_4 \times P_x) + (2 \times Y_4 \times P_2) + P_o \geq -(X_4 \times R_m)$
case 4:	$(X_4 \times P_x) + (2 \times Y_5 \times P_e) + (2 \times P_o) \geq -(X_4 \times R_m)$
case 5:	$(X_4 \times P_x) + (2 \times Y_5 \times P_e) + P_o \geq -(X_4 \times R_m)$
case 6:	$(X_4 \times P_x) + P_o \geq -(X_4 \times R_m)$
case 7:	$P_x \geq -R_m$
case 8:	$(X_4 \times P_x) + (2 \times Y_4 \times P_e) + P_o \geq -(X_4 \times R_m)$
case 9:	$(X_4 \times P_x) + (2 \times Y_5 \times P_e) + P_o \geq -(X_4 \times R_m)$
case 10:	$P_o \geq 0$
case 11:	$(2 \times Y_4 \times P_e) + P_o \geq 0$
case 12:	$(2 \times Y_5 \times P_e) + P_o \geq 0$

Figure 10.2: Possible arrangements of MEMs in Equation 7.1a.

10.9 Ukkonen and GeneMyer implementation details

In order to compute the performance and accuracy of the Ukkonen and GeneMyer algorithms the methods below are used. Note that both Ukkonen and GeneMyer in their original format only return the number of edits but not the alignment path. Since the alignment scores for the alignments produced by these algorithms are needed for comparison with the optimal alignment scores produced by Smith-Waterman, modified versions of these algorithms were used in our evaluation. All Linux scripts, C source codes are available in the MEM-Align packages.

UKKonen: the SNAP short read mapper implements a modified version of Ukkonen that returns the alignment path in a CIGAR string format (refer to the SAM file format specification for details). We take this implementation of Ukkonen for evaluation. The CIGAR string is then processed by a program to compute the alignment score. Note that in the CIGAR string produced by this implementation of Ukkonen matches and mismatches are represented with different symbols ("=", "X" respectively); thus the alignment score can be computed from the CIGAR string. The execution time for computing alignment scores from CIGAR strings is not included in execution time of Ukkonen.

GeneMyer We used an implementation of GeneMyer algorithm from the SeqAN package. This implementation is also used in [24] for evaluation purposes.

Note that GeneMyer is implemented as a global alignment in the SeqAN package and the reported alignment score is a global alignment score which cannot be compared against the optimal local alignment score produced by Smith-Waterman. In order to compute the local alignment score for the alignment produced by GeneMyer we print the alignment and then a program compute local alignment score for the produced alignment. Since printing alignment is a time-consuming file operation, for a fair comparison, we first processed the input sequence pairs without printing out the alignment and recorded the execution time. Then we processed the input once again to print out the alignment.

10.10 Detailed Experimental Results

Table 10.1: MEM extraction statistics for all datasets and differing parameters. ET: Execution Time (sec). AM: Average number of extracted MEMs per sequence pair. NBP: Number of sequence pairs with more than TM MEMs (see 6.5)

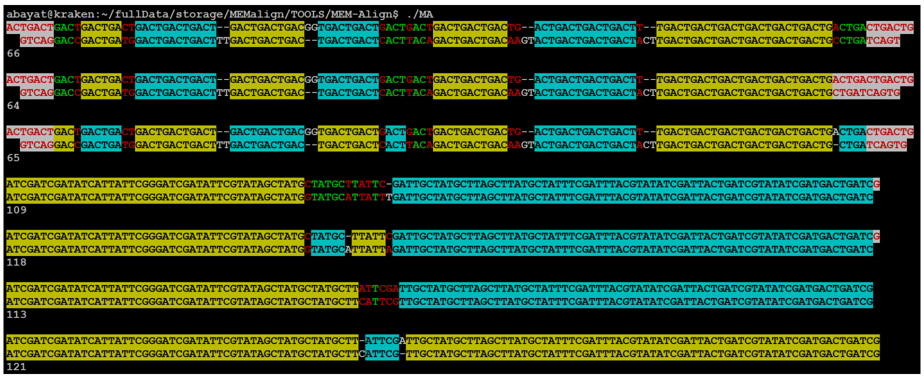
gl	sl	TM	DSL			DSH		
			ET	AM	BPM	ET	AM	BPM
124	0	∞	159.4	3031.2	0	174.9	3034.0	0
124	0	100	12.3	0.0	2000000	12.7	0.0	2000000
124	4	∞	113.9	76.7	0	112.5	78.2	0
5	0	∞	10.9	244.8	0	11.5	249.6	0
5	4	∞	7.7	9.5	0	7.7	13.2	0
5	4	100	7.6	9.5	0	7.1	13.2	0
5	2	100	8.7	73.3	87465	8.5	76.7	118690
10	4	100	12.3	16.5	440	12.1	19.8	366
gl	sl	TM	DLL			DLH		
			ET	AM	BPM	ET	AM	BPM
499	0	∞	2091.6	48101.8	0	2200.0	48103.6	0
499	0	300	78.6	0.0	2000000	79.4	0.0	2000000
499	4	∞	1720.5	1105.0	0	1668.7	1090.6	0
10	0	∞	70.0	1959.6	0	69.3	1977.9	0
10	4	∞	44.0	65.9	0	41.9	79.9	0
10	4	300	50.8	65.7	892	51.3	79.8	771
10	2	300	29.6	278.3	1999875	29.1	288.7	1999989
20	4	300	89.6	115.4	8900	89.4	128.0	9022

Table 10.2: Extraction and sorting execution time

Dataset	gl	sl	Extraction time	Sort time	Total time	% of Sort
DSL	10.0	2.0	12.6	5.6	18.2	30.8
	5.0	4.0	8.3	0.4	8.7	4.6
DLL	20.0	2.0	83.3	38.2	121.5	31.4
	10.0	4.0	42.1	7.1	49.2	14.4

Table 10.3: Alignment performance, accuracy and statistics. SW indicates the time spent on Smith-Waterman to align those sequences which are bypassed by MEM-Align. Bypassed is the number of alignment bypassed. NSOA represents number of suboptimal alignment produce by MEM-Align. C1 is the number of expected S_i^j computation. C2 is the number of S_i^j computation after optimisation in Section 6.1, C3 is number of S_i^j after considering TD. C4 is number of time that sequential compare between sequences is needed (see Section 6.4). C5 is actual number of time sequential compare is done. Note that by computing minimum P_i^j we could avoid comparing sequences(see Section 6.4).

Dataset Code	Parameters					Execution Time (sec)				Number of pairs		Runtime Statistics for 1000 sequence pairs				
	gl	sl	TM	TD	Po	MEM Align	SW	Total	% SW	Bypassed	NSOA	C1	C2	C3	C4	C5
DSL	5	4	100	20	6	11.5	0.0	11.5	0.0	0	240	55314.6	29876.9	15006.7	10083.3	2026.4
	5	2	100	20	6	66.9	2.7	69.6	3.9	87465	141	2718398.3	685229.7	540755.4	414471.4	24312.5
	10	4	100	20	6	20.7	0.0	20.8	0.2	440	170	168815.1	96222.5	47637.9	32741.9	4657.2
	5	4	100	10	6	11.1	0.0	11.1	0.0	0	247	55314.6	29876.9	10268.8	5345.3	984.8
	5	4	100	30	6	11.3	0.0	11.3	0.0	0	240	55314.6	29876.9	18615.6	13692.1	2826.2
	5	4	500	20	6	11.9	0.0	11.9	0.0	0	240	55314.6	29876.9	15006.7	10083.3	2026.4
	5	4	100	20	9	10.5	0.0	10.5	0.0	0	189	55314.6	29876.9	15006.7	10083.3	1886.5
DSH	5	4	100	20	6	13.3	0.0	13.3	0.0	0	13458	94616.1	47836.6	26807.6	16598.3	4973.1
	5	2	100	20	6	71.1	4.0	75.1	5.3	118690	6228	2963108.2	750217.9	598808.4	456433.9	43519.2
	10	4	100	20	6	23.8	0.0	23.8	0.1	366	8199	226112.2	124195.0	65414.3	43492.3	9410.0
	5	4	100	10	6	12.2	0.0	12.2	0.0	0	19233	94616.1	47836.6	19441.7	9232.4	3286.2
	5	4	100	30	6	13.5	0.0	13.5	0.0	0	13446	94616.1	47836.6	32199.0	21989.8	5722.6
	5	4	500	20	6	12.5	0.0	12.5	0.0	0	13458	94616.1	47836.6	26807.6	16598.3	4973.1
	5	4	100	20	9	12.5	0.0	12.5	0.0	0	9727	94616.1	47836.6	26807.6	16598.3	4708.1
DLL	10	4	300	20	6	108.6	0.7	109.3	0.6	892	598	2361465.1	946769.7	220871.1	155623.3	39046.7
	10	2	300	20	6	29.4	712.5	741.9	96.0	1999875	63	38820864.0	5051392.0	2380800.0	1585152.0	171008.0
	20	4	300	20	6	263.4	3.7	267.1	1.4	8900	596	7139541.6	3098397.3	668244.4	483218.0	128956.6
	10	4	300	10	6	102.9	0.7	103.6	0.7	892	698	2361465.1	946769.7	143434.9	78187.2	14198.3
	10	4	300	30	6	115.7	0.7	116.4	0.6	892	598	2361465.1	946769.7	285578.8	220331.0	64921.0
	10	4	1000	20	6	100.6	0.0	100.6	0.0	0	598	2389791.8	949235.9	222717.2	156942.0	39250.6
	10	4	300	20	9	107.7	0.8	108.4	0.7	892	419	2361465.1	946769.7	220871.1	155623.3	38282.5
DLH	10	4	300	20	6	126.5	0.2	126.7	0.2	771	53202	3359559.8	1159348.3	298648.2	203998.7	59722.7
	10	2	300	20	6	31.5	636.3	667.8	95.3	1999989	63	41576727.3	5189818.2	2455272.7	1524363.6	104727.3
	20	4	300	20	6	290.4	2.8	293.2	1.0	9022	47063	8612853.2	3429913.4	779279.9	556304.0	135951.1
	10	4	300	10	6	120.8	0.5	121.3	0.4	771	117909	3359559.8	1159348.3	200085.0	105435.5	34640.3
	10	4	300	30	6	130.3	0.6	130.9	0.4	771	53031	3359559.8	1159348.3	378930.1	284280.5	71699.3
	10	4	1000	20	6	118.9	0.0	118.9	0.0	0	53218	3382775.5	1161422.5	300185.7	205112.4	59841.7
	10	4	300	20	9	127.0	0.6	127.5	0.4	771	38783	3359559.8	1159348.3	298648.2	203998.7	59123.8



```
abayat@kraken:~/fullData/storage/MEMalign/TOOLS/MEM-Align$ ./MA
ACTGACTGACTGACTGACTGACTGACT--GACTGACTGACGGTGACTGACTGACTGACTGACTGACTGACTG--ACTGACTGACTGACTT--TGACTGACTGACTGACTGACTGACTGACTGACTGACTG
  GTCAGGACCGACTGATGGACTGACTGACTTTGACTGACTGAC--TGACTGACTCACTTACAGACTGACTGACAAGTACTGACTGACTGACTACTTGACTGACTGACTGACTGACTGACTGCCTGATCAGT
66

ACTGACTGACTGACTGACTGACTGACT--GACTGACTGACGGTGACTGACTGACTGACTGACTGACTGACTG--ACTGACTGACTGACTT--TGACTGACTGACTGACTGACTGACTGACTGACTGACTG
  GTCAGGACCGACTGATGGACTGACTGACTTTGACTGACTGAC--TGACTGACTCACTTACAGACTGACTGACAAGTACTGACTGACTGACTACTTGACTGACTGACTGACTGACTGACTGCTGATCAGTG
64

ACTGACTGACTGACTGACTGACTGACT--GACTGACTGACGGTGACTGACTGACTGACTGACTGACTGACTG--ACTGACTGACTGACTT--TGACTGACTGACTGACTGACTGACTGACTGACTGACTG
  GTCAGGACCGACTGATGGACTGACTGACTTTGACTGACTGAC--TGACTGACTCACTTACAGACTGACTGACAAGTACTGACTGACTGACTACTTGACTGACTGACTGACTGACTGACTG-CTGATCAGTG
65

ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGCTATGCTTATTC-GATTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATCG
ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGGTATGCATTATTTGATTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATC
109

ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGCTATGC-TTATTCGATTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATCG
ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGGTATGCATTATTAGATTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATC
118

ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGCTATGCTTATTCGATTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATCG
ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGCTATGCTTCATTCGTTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATCG
113

ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGCTATGCTT-ATTCGATTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATCG
ATCGATCGATATCATTATTCGGGATCGATATTCGTATAGCTATGCTATGCTTCATTCG-TTGCTATGCTTAGCTTATGCTATTTCGATTTACGTATATCGATTACTGATCGTATATCGATGACTGATCG
121
```

Figure 10.3: Graphical representation of alignment using MEM-Align. Blue and Yellow are MEMs. Red shows mismatches. Green represent recovered short MEMs. From top to down the second, fourth and sixth alignment are not optimal and the third, fifth and seventh alignment show the optimal alignment of same sequences respectively