2015 IEEE International Parallel and Distributed Processing Symposium Workshops

Generalised Implementation for Fixed-Length Approximate String Matching under Hamming Distance & Applications

Solon P. Pissis and Ahmad Retha
Department of Informatics, King's College London, London, UK
{*solon.pissis,ahmad.retha*}*@kcl.ac.uk*

***Abstract*—Approximate string matching is the problem of finding all factors of a text t of length n with a distance at most k from a pattern x of length $m \leq n$. Fixed-length approximate string matching is the problem of finding all factors of a text t of length n with a distance at most k from *any* factor of length h of a pattern x of length $m \leq n$, where $h \leq m$. It is thus a generalisation of approximate string matching and has numerous direct applications in molecular biology—motif extraction and circular sequence alignment, to name a few.**

MaxShiftM is a bit-parallel algorithm for fixed-length approximate string matching under the Hamming distance model with time complexity $\mathcal{O}(m\lceil h/w \rceil n)$, where w is the size of the computer word (Crochemore *et al.*, 2010). An implementation of this algorithm is straightforward as long as the maximal length of alignments is less than or equal to w. In this article, our contribution is twofold: first, we propose a generalised implementation of MaxShiftM, that is, for any given $h \leq m$, under the Hamming distance model; and second, we show how our implementation can be used to improve the accuracy and efficiency of multiple circular sequence alignment in terms of the inferred likelihood-based phylogenies.

***Keywords*-algorithms on strings; approximate string matching; dynamic programming.**

I. Introduction

Various algorithms exist to tackle the problem of finding an exact or similar string *pattern* in a given string *text*. The topic contains many competing algorithms for solving specific problems in particular branches of scientific research [17]. In the field of molecular biology, *string matching* algorithms are used for extracting motifs or regions of interest [18] or for the alignment of short reads to longer DNA or protein sequences [24]. This is important for solving problems such as gene classification, protein function identification, mutation discovery, and genetic sequencing.

The current state of art in the field present a number of applications with algorithms that are general-purpose or directly applicable in molecular biology. The general computing problem these algorithms attempt to solve is known as *exact string matching* and aims to determine whether or not a string of letters is present inside another string. Furthermore, this problem expands into the realms of *approximate string matching*; approximate matching allows for a limited number of *errors* or, dually, *edit operations* needed for the searched pattern to have a match in the text.

One of the most commonly used error models is the so called *edit distance* model, which uses substitution, deletion and insertion of letters in both strings. Each of these different operations can have a different edit *cost* or the cost may depend on the letters involved; in this case we speak of the *general edit distance* model. Otherwise, if all the operations have a unit cost, we speak of simple edit distance or just edit distance. In the latter case we simply seek for the minimum number of operations to transform the one string into the other. A further simplification of this model is the *Hamming distance* model and this variant only allows for substitutions.

DNA errors occur at a steady rate with the average single nucleotide substitution mutation rate in humans estimated to be 1.20×10^{-8} per nucleotide per generation [13]. Apart from single-nucleotide substitution, there is also nucleotide deletion and insertion rates but these are significantly lower. Although more specialised—but computationally more expensive—error models exist for molecular sequences [11], the edit and Hamming distance models are still extensively used as a reasonable accuracy-efficiency trade-off.

Two approaches to devising search algorithms can be found in the literature, termed *off-line* and *on-line*. Off-line algorithms have a preprocessing stage where they take the input datasets and generate exhaustive intermediate indexes which can then be searched. Off-line algorithms presuppose the search datasets remain mostly static and index-storage space is abounding. Often, indexing time is of secondary importance to search time and this makes it impractical for our purposes. On-line search algorithms work on changing and expanding datasets and run on-the-fly with little or no preprocessing. On-line search algorithms are better suited for immediate and on-demand searching of variable datasets. In this work we consider only on-line approaches.

With edit distance, the searching problem is known as *approximate string matching with k-errors*. The first algorithm for solving this problem has been rediscovered many times in the past in different areas [17]. However this algorithm computed the edit distance, and it was converted into a search algorithm only in 1980 by Sellers [22]. Sellers algorithm requires time $\mathcal{O}(mn)$, where m is the length of the pattern and n is the length of the text. The major theoretical and practical achievements are $\mathcal{O}(kn)$-time algorithms,

/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.106

presented in [8], [9], [26], where k is the maximum edit distance allowed. An intensive study on the question is by Navarro [17].

A thread of practice-oriented results exploits the hardware word-level parallelism of operations. It takes advantage of the intrinsic parallelism of the word-level operations inside a computer word. By using this fact cleverly, the number of operations that a bit-parallel algorithm performs can be cut down by a factor of at most w, where w is the size of the computer word. In order to relate the behavior of bit-parallel algorithms to other work, it is normally assumed that $w = \Theta(\log n)$, as dictated by the RAM model. Wu and Manber, in [29], gave an $\mathcal{O}(k\lceil m/w\rceil n)$-time algorithm for the k-errors problem by simulating the non-deterministic automaton [25] using word-level parallelism. Baeza-Yates and Navarro, in [1], gave an $\mathcal{O}(\lceil km/w\rceil n)$-time variation of the Wu-Manber algorithm, implying $\mathcal{O}(n)$ performance when $km = \mathcal{O}(w)$. In 1994, Wright [28] presented an $\mathcal{O}(m\log(|\Sigma|)n/w)$-time algorithm, where $|\Sigma|$ is the size of the input alphabet Σ. This was the first work using word-level parallelism on the dynamic programming matrix. Myers, in [16], gave an $\mathcal{O}(\lceil m/w\rceil n)$-time algorithm using word-level parallelism for computing the dynamic programming matrix more efficiently. Another general solution based on existing algorithms can be found in [5].

A generalisation of this problem—instead of searching for all matches of the pattern—is to search for all matches of *any factor* of some fixed length $h \leq m$ of the pattern. This problem is known as *fixed-length approximate string matching* and it was introduced by Iliopoulos *et al.* in [12]. Crochemore *et al.* devised MaxShiftM [6], a bit-parallel algorithm for fixed-length approximate string matching under the Hamming distance model with time complexity $\mathcal{O}(m\lceil h/w\rceil n)$. MaxShiftM is currently the core computational task of at least two applications in molecular biology:

- MoTeX: a tool for single and structured motif extraction from large-scale datasets [18], [19]. With MoTeX, MaxShiftM is used to detect overrepresented motifs in the same or related genetic sequences. Regulatory regions such as promoter and enhancer binding sites can be found surrounding genes in DNA. They are the attachment points for transcription factors and are mostly conserved but can contain substitutions of non-conserved bases and a small number of gaps. MaxShiftM is adept at finding regulatory regions as it can find these binding sites, short patterns, in long strings by using fixed-length approximate string matching.
- BEAR: a tool for multiple circular sequence alignment [2]. With BEAR, MaxShiftM is used to identify common approximate factors between circular sequences in order to find a sufficiently good rotation of the involved sequences. This is a crucial step because sequences of circular structure may be cut at arbitrary positions to be sequenced so one copy of a genetic code will not be aligned with another. BEAR uses MaxShiftM to search for a sufficiently good rotation for each of the sequences against one another, so that when they are passed onto a conventional multiple sequence alignment tool, it will be able to provide a better multiple alignment.

MaxShiftM operates by maintaining a variant of the traditional dynamic programming matrix D, denoted by D′, through iteratively comparing the letters in text t against the letters in the pattern x to provide a representation for solving the fixed-length approximate string matching problem. MaxShiftM maintains the alignments information in matrix D′ via word-level `shift-or` operations. Edit operations are stored in individual bits of a computer word and at the end of this computation, the sum of these operations denote the alignment cost; this cost can be determined by obtaining the sum of bits set on in the computer word via a word-level `popcount` operation. The resulting matrix D′ holds the binary encoding for obtaining the optimal alignment between any factor of length h of x and some factor of t.

MaxShiftM requires time $\mathcal{O}(mn)$ under the assumption that the maximal length of the stored alignments is less than or equal to w. This ensures that each cell of the dynamic programming matrix can be computed in constant time. For instance, a naïve implementation running on a regular 64-bit computer, cannot handle alignments for $h > 64$. Hence it is well understood that, while this may be sufficient for short alignments, it is not general enough for certain applications.

Our Contribution

In this article, we present MW-MaxShiftM, a robust and generalised implementation of MaxShiftM which overcomes the limitations of a naïve implementation of the algorithm, so that it no longer requires that the maximal length of the alignments is less than or equal to w ($h \leq w$). With MW-MaxShiftM, the fixed length h can be set as long as length m of pattern x ($h \leq m$).

MW-MaxShiftM carries out a similar process as MaxShiftM but stores the alignments across $\lceil h/w\rceil$ words per cell giving it an overall time complexity of $\mathcal{O}(m\lceil h/w\rceil n)$. In particular our contribution is twofold:

1) We provide the full details of a generalised implementation of the word-level operations used for the maintenance of the dynamic programming matrix.
2) We show how our implementation can be used to improve the efficiency and accuracy of multiple circular sequence alignment in terms of the inferred likelihood-based phylogenies.

II. Definitions and Notation

In order to provide an overview of our results and algorithms, we begin with a few definitions. We think of a *string*

x of *length* m as an array $x[0\,..\,m-1]$, where every $x[i]$, $0 \leq i < m$, is a *letter* drawn from some fixed *alphabet* Σ of size σ. The *empty string* of length 0 is denoted by ε. A string x is a *factor* of a string t if there exist two strings u and v, such that $t = uxv$. Consider the strings x, t, u, and v, such that $t = uxv$. If $u = \varepsilon$, then x is a *prefix* of t. If $v = \varepsilon$, then x is a *suffix* of t.

Let x be a non-empty string of length m and t be a string. We say that there exists an *occurrence* of x in t, or, more simply, that x *occurs in* t, when x is a factor of t. Every occurrence of x can be characterised by a position in t. Thus we say that x occurs at the *starting position* i in t when $t[i\,..\,i+m-1] = x$. We say that x occurs at the *ending position* i in t when $t[i-m+1\,..\,i] = x$.

Given a string x and a string y, the *edit distance*, denoted by $\delta_E(x, y)$, is defined as the minimum total cost of operations required to transform one string into the other. For simplicity, we only count the number of edit operations, considering the cost of each to be 1. The allowed edit operations are as follows:

- *Insertion*: insert a letter in y, not present in x; $(\varepsilon, b),\ b \neq \varepsilon$
- *Deletion*: delete a letter in y, present in x; $(a, \varepsilon),\ a \neq \varepsilon$
- *Substitution*: replace a letter in y with a letter in x; $(a, b),\ a \neq b, \texttt{and}\ a, b \neq \varepsilon$.

We write $x \equiv_k^E y$ if the edit distance between x and y is at most k. Equivalently, if $x \equiv_k^E y$, we say that x and y have at most k *errors*.

Given a string x and a string y both of length m, the *Hamming distance*, denoted by $\delta_H(x, y)$, is the number of positions i, $0 \leq i < m$, such that $x[i] \neq y[i]$. Given an integer $k > 0$, we write $x \equiv_k^H y$ if the Hamming distance between x and y is at most k. Equivalently, if $x \equiv_k^H y$, we say that x and y have at most k *mismatches*.

We say that there exists an *occurrence* of a non-empty string x of length m in a string t of length $n \geq m$ with at most k mismatches, or, more simply, that x *occurs in* t with at most k mismatches, when $u \equiv_k^H x$ and u is a factor of t. We refer to the *standard dynamic programming matrix* of x and t as the matrix defined by $\mathsf{D}[i, 0] = k + 1,\ 0 < i \leq m,\ \mathsf{D}[0, j] = 0,\ 0 \leq j \leq n$

$$\mathsf{D}[i,j] = \begin{cases} \mathsf{D}[i-1, j-1] & : \quad x[i] = t[j] \\ \mathsf{D}[i-1, j-1] + 1 & : \quad x[i] \neq t[j] \end{cases}$$

for all $0 < i \leq m, 0 < j \leq n$.
Similarly we refer to the *standard dynamic programming algorithm* as the algorithm to compute the Hamming distance between x and any factor of t through the above recurrence in time $\mathcal{O}(mn)$. That is, $\mathsf{D}[m, j] \leq k$ denotes an occurrence of x at the ending position $j-1$ in t with at most k mismatches.

A circular string of length m can be viewed as a traditional linear string which has the left- and right-most letters wrapped around and stuck together in some way. Under this notion, the same circular string can be seen as m different linear strings, which would all be considered equivalent. Given a string x of length m, we denote by $x^i = x[i\,..\,m-1]x[0\,..\,i-1]$, $0 < i < m$, the i-th *rotation* of x and $x^0 = x$. Consider, for instance, the string $x = x^0 = \texttt{abababbc}$; this string has the following rotations: $x^1 = \texttt{bababbca}$, $x^2 = \texttt{ababbcab}$, $x^3 = \texttt{babbcaba}$, $x^4 = \texttt{abbcabab}$, $x^5 = \texttt{bbcababa}$, $x^6 = \texttt{bcababab}$, $x^7 = \texttt{cababab}$.

In this article, we consider the following problem.

FIXEDLENGTHAPPROXIMATESTRINGMATCHING
Input: a pattern x of length m, a text t of length $n \geq m$, an integer $h \leq m$, and an integer threshold $k < h$
Output: all factors u of t such that $u \equiv_k^H v$, where v is any factor of length h of x

III. ALGORITHM MAXSHIFTM

Let $\mathsf{D}'[0\,..\,m, 0\,..\,n]$ be a matrix, where $\mathsf{D}'[i, j]$ contains the Hamming distance between factor $t[\max\{0, j-h\}\,..\,j-1]$ of t and factor $x[\max\{0, i-h\}\,..\,i-1]$ of x, for all $1 \leq j \leq n$, $1 \leq i \leq m$. This matrix can be obtained through a straightforward $\mathcal{O}(hmn)$-time algorithm by constructing matrices $\mathsf{D}^s[0\,..\,h, 0\,..\,n]$, for all $0 \leq s \leq m-h$, where $\mathsf{D}^s[i, j]$ is the Hamming distance between factor $t[j-h\,..\,j-1]$ and the prefix of length i of $x[s\,..\,s+h-1]$. We obtain D' by collating D^0 and the last row of D^s, for all $0 \leq s \leq m-h$. Matrices D^s can be obtained using the standard dynamic programming algorithm. We say that $x[\max\{0, i-h\}\,..\,i-1]$ occurs in t ending at $t[j-1]$ with k mismatches *iff* $\mathsf{D}'[i, j] \leq k$, for all $1 \leq i \leq m$, $1 \leq j \leq n$.

Let $x = \texttt{CAAACCTTT}$, $t = \texttt{CGAAAGTAT}$, and $h = 3$. Matrix D' is given below.

		0	1	2	3	4	5	6	7	8	9
		ϵ	C	G	A	A	A	G	T	A	T
0	ϵ	0	0	0	0	0	0	0	0	0	0
1	C	1	0	1	1	1	1	1	1	1	1
2	A	2	2	1	1	1	1	2	2	1	2
3	A	3	3	3	1	1	1	2	3	2	2
4	A	3	3	3	2	1	0	1	2	2	2
5	C	3	2	3	3	2	1	1	2	3	2
6	C	3	2	2	3	3	2	2	2	3	3
7	T	3	3	2	2	3	3	3	2	3	2
8	T	3	3	3	2	3	3	3	2	2	2
9	T	3	3	3	3	3	3	3	2	2	1

The algorithm finds $\mathsf{D}'[4, 5] = 0$, since $\delta_H(x[1\,..\,3], t[2\,..\,4]) = 0$. The algorithm finds $\mathsf{D}'[9, 9] = 1$, since $\delta_H(x[6\,..\,8], t[6\,..\,8]) = 1$.

Crochemore *et al.* devised MaxShiftM [6], a bit-parallel algorithm with time complexity $\mathcal{O}(m\lceil h/w \rceil n)$, where w is the size of the computer word. By using word-level parallelism, MaxShiftM can compute matrix D' efficiently.

Let $\mathsf{B}[i,j] = b_{h-1} \dots b_0$, a bit-vector matrix holding the binary encoding for obtaining the Hamming distance between the factor of the text $t[j-h\,..\,j-1]$ and $x[i-h\,..\,i-1]$ with a total of $\mathsf{D}'[i,j]$ mismatches. The maintenance of matrix B is done via operations defined as follows:

- `shift(`v`)`: an operation that, given a bit-vector v, shifts the bits of v one position to the left, and returns the resulting bit-vector.
- `leftmostbit(`v`)`: an operation that, given a bit-vector v, returns the leftmost bit of v.
- `shiftc(`v`)`: same as `shift`, but also truncates the leftmost bit of v.
- `|`: the bitwise `OR` operator.
- `&`: the bitwise `AND` operator.
- `popcount(`v`)`: an operation that, given a bit-vector v, returns the number of 1's in v.

Algorithm *MaxShiftM*(x, m, t, n, h)
  $\mathsf{D}'[0\,..\,m, 0\,..\,n] \leftarrow 0$;
  $\mathsf{B}[0\,..\,m, 0\,..\,n] \leftarrow 0$;
  foreach $i \in \{1, m\}$ **do**
    $\mathsf{B}[i, 0] \leftarrow \min(i, h)$ 1's;
    foreach $j \in \{1, n\}$ **do**
      $\mathsf{B}[i,j] \leftarrow$ `shiftc`$(\mathsf{B}[i-1, j-1]) \mid \delta_H(x[i-1], t[j-1])$;
      $\mathsf{D}'[i,j] \leftarrow$ `popcount(`$\mathsf{B}[i,j]$`)`;

The algorithm requires constant time for computing each cell $\mathsf{B}[i,j]$ by using the aforementioned operations, assuming that $\lceil h/w \rceil = \mathcal{O}(1)$. In the general case, it requires time $\mathcal{O}(\lceil h/w \rceil)$. The space complexity can be reduced to only $\mathcal{O}(m\lceil h/w \rceil)$ since each column of D' only depends on the immediately preceding column.

Theorem 1 ([6]): Given a string x of length m, a string t of length n, an integer h, and the size of the computer word w, algorithm MaxShiftM requires time $\mathcal{O}(m\lceil h/w \rceil n)$ to solve the FIXEDLENGTHAPPROXIMATESTRINGMATCHING problem.

IV. IMPLEMENTATION

In this section we provide the full details of a generalised implementation. In particular, we provide an implementation for the operations used for the maintenance of the dynamic programming matrix. For the rest of this section we denote by:

- v an array of bit-vectors used to represent a cell of the dynamic programming matrix;
- $s := \lceil h/w \rceil$ the size of array v;
- $\ell := h$ the length of the alignment (bit-vector) stored in v.

The `mw-shift` function is an implementation of operation `shift` on multiple words. The `mw-shift` function performs a bitwise left-shift operation on v. It starts from right to left, performing a bitwise left-shift on the rightmost element in v, and moves left the array towards the leftmost element performing a bitwise left-shift on each element as it goes along. It checks if a bit needs to be carried onto the next element in the array and carries these bits along at each step. It requires time $\mathcal{O}(s)$.

Function *mw-shift*(v, s, w)
  $a \leftarrow 1 \ll (w-1)$;
  $b \leftarrow 0$;
  for $i \leftarrow s-1$ **downto** 0 **do**
    $c \leftarrow v[i]$;
    $v[i] \leftarrow (v[i] \ll 1) \mid b$;
    if $(c \& a) > 0$ **then**
      $b \leftarrow 1$; /* we need to carry a bit */
    else
      $b \leftarrow 0$; /* we do not need to carry a bit */
  return v;

An example of this function is shown below.

Example 1: `mw-shift` for $h = 32$ and $w = 8$

v_0	v_1	v_2	v_3
10000011	10000010	10011101	01111111
00000111	00000101	00111010	11111110

The `mw-shiftc` function is an implementation of operation `shiftc` on multiple words. The `mw-shiftc` function is almost identical to the `mw-shift` function, except that it truncates the most significant bit in the most significant element after performing the shift. The location of this bit is maintained by ℓ and it represents the current length of the alignment. The function uses a bit-mask (variable y below) which is calculated based on ℓ and w to truncate the most significant bit in order to stop more errors that are no longer part of the alignment being counted. It requires time $\mathcal{O}(s)$.

Function *mw-shiftc*(v, s, w, ℓ)
  `mw-shift`(v, s, w); /* we first shift and then truncate */
  $j \leftarrow \ell \bmod w$;
  $y \leftarrow j$ 1's;
  $i \leftarrow s - \lceil \ell/w \rceil$;
  $v[i] \leftarrow v[i] \& y$;
  return v;

An example of this function is shown below.

Example 2: `mw-shiftc` for $\ell = 20$ and $w = 8$			
Operation	v_0	v_1	v_2
`mw-shift`	00001011	10011101	01111111
$v[0]$&1111	00010111	00111010	11111110
Result	00000111	00111010	11111110

The `mw-popcount` function is an implementation of operation `popcount` on multiple words. The `mw-popcount` function performs a `popcount` on all the elements in v. It returns the total count of set bits set on in all the elements of the array. Each "1" represents an error in the alignment. Function `mw-popcount` requires time $\mathcal{O}(s)$ assuming that operation `popcount` is implemented in constant time.

Function *mw-popcount*(v, s)
  $c \leftarrow 0$;
  foreach $i \in \{0, s-1\}$ **do**
    $c \leftarrow c +$ `popcount(v[s])`;
  return c;

An example of this function is shown below.

Example 3: `mw-popcount` for $h = 16$ and $w = 8$		
$\mathsf{D}'[i,j]$	v_1	v_2
8	10011010	00111001

We are now in a position to present the implementation of algorithm MaxShiftM on multiple words. The MW-MaxShiftM implementation takes six arguments: x is the pattern string and m its length; t is the text string and n its length; h is the length of the factor we are looking for; and w is the size of the computer word.

Algorithm *MW-MaxShiftM*(x, m, t, n, h, w)
  $s \leftarrow \lceil h/w \rceil$;
  $\mathsf{D}'[0\,..\,m, 0\,..\,n, 0\,..\,s-1] \leftarrow 0$;
  $\mathsf{B}[0\,..\,m, 0\,..\,n, 0\,..\,s-1] \leftarrow 0$;
  foreach $i \in \{1, m\}$ **do**
    $\mathsf{B}[i, 0] \leftarrow \min(i, h)$ 1's;
    foreach $j \in \{1, n\}$ **do**
      $\mathsf{B}[i,j] \leftarrow$ `mw-shiftc`$(\mathsf{B}[i-1, j-1], s, w, h) \mid \delta_H(x[i-1], t[j-1])$;
      $\mathsf{D}'[i,j] \leftarrow$ `mw-popcount`$(\mathsf{B}[i,j], s)$;

Overall, our implementation requires constant time for computing each cell $\mathsf{D}'[i,j]$ by using word-level operations, assuming that $\lceil h/w \rceil = \mathcal{O}(1)$. In the general case, it requires time $\mathcal{O}(\lceil h/w \rceil)$. Trivially, the space complexity can be reduced to only $\mathcal{O}(m\lceil h/w \rceil)$ since each column of D' only depends on the immediately preceding column.

V. Experimental Results

- Project name: MW-MaxShift
- Project home page: http://github.com/webmasterar/mw_maxshift
- Operating system: GNU/Linux
- Programming language: C
- Other requirements: compiler `gcc` version 4.6.3 or higher
- License: GNU GPL
- Any restrictions to use by non-academics: license needed

The experiments were conducted on a Desktop PC using one core of a 64-bit Intel 2.8GHz Core-I7 processor with 8GB of RAM under 64-bit GNU/Linux.

Experiment 1: Efficiency

As the first experiment we tested the efficiency of our implementation. As an input dataset we used a pattern sequence of length $m = 1,536$ and a text sequence of length $n = 2,000$, both over the DNA alphabet; notice that the time complexity of MaxShiftM does not depend on the alphabet's size. To measure the efficiency of our implementation we used different values for the fixed factor length h, $8 \leq h \leq m$. The results are shown in Figure 1. The shape of the curve fully confirms the theoretical findings. For $h \leq 64$, we make use of a single computer word as the maximal length of any alignment is bounded by $h = 64$. Since the x axis (fixed length h) is on *logarithmic* scale this implies that the execution time grows *linearly* in h thereafter. This linear growth is represented by the $\lceil h/w \rceil$ factor in the time complexity of MaxShiftM (see Theorem 1).

Experiment 2: Application I (Synthetic Data)

Circular DNA can be found in viruses, archaea, and bacteria as plasmids and in the mitochondria and plastids of eukaryotic cells. Hence algorithms on circular sequences are important in the analysis of organisms with such genetic structures [3]. An application of circular sequences has been in the context of reconstructing phylogenies using Mitochondrial DNA (MtDNA) [10], [7], [27]. MtDNA is generally conserved from parent to offspring, and so it can be used as an indicator of evolutionary relationships among species. The absence of recombination in these sequences allows it to be used as a simple test of phylogenetic evolution, and the high mutation rate leads to a powerful discriminative feature. However, when sequencing a DNA molecule, the position where a circular genome starts can be totally arbitrary. Due to this arbitrary definition, using conventional tools to align such sequences could yield an incorrectly high genetic distance between closely-related species. For instance, the linearised human (NC_001807) and chimpanzee (NC_001643) MtDNA sequences do not start in the same region. Their pairwise sequence alignment using EMBOSS Needle [20] gives a similarity of 85.1%

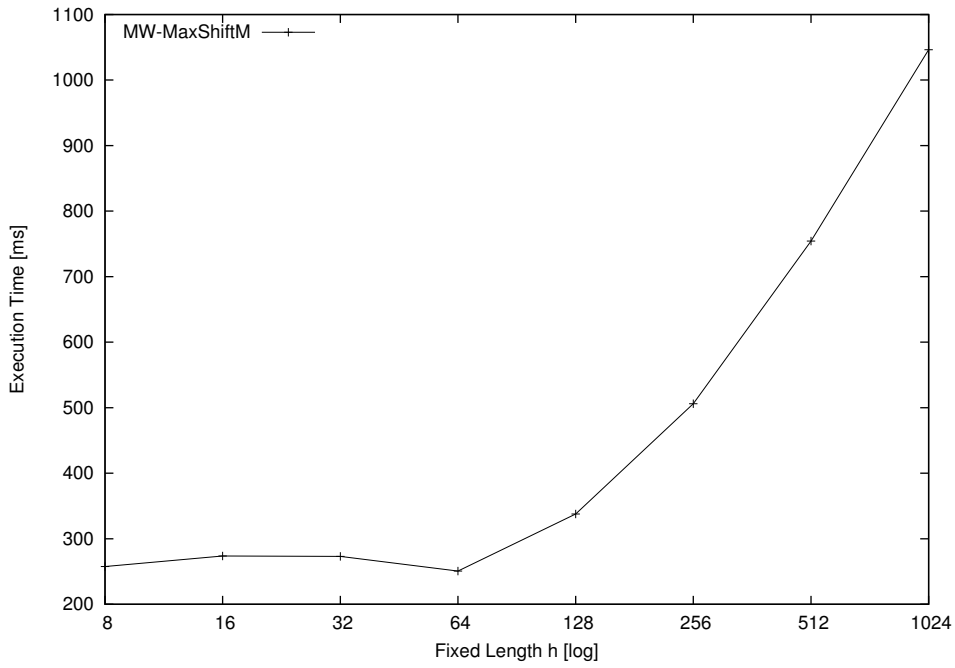

Figure 1: Fixed length h versus elapsed time for $m = 1,536$ and $n = 2,000$ using MW-MaxShiftM

and consists of 1195 gaps; taking different rotations of these sequences into account yields an alignment with a similarity of 91% and only 77 gaps.

Approximate circular string matching is the problem of finding all factors of t that are at a distance at most k from x or from *any* of its rotations [3], [4]. Consider comparing x and t under the Hamming distance model. We can apply MaxShiftM$(x', 2m, t, n, h)$ with $x' = xx$ and $h := m$. Notice that cell $\mathsf{D}'[p, q]$, say $\mathsf{D}'[p, q] = e$, denotes that factor $x'[p - h\,..\,p - 1]$ matches factor $t[q - h\,..\,q - 1]$ with e mismatches. Hence by setting $h := m$ we can report all factors of t that are at a Hamming distance at most k from x or from any of its rotations. Notice that $x'[p-m\,..\,p-1]$, $p - m \geq 0$, denotes rotation x^{p-m} of x.

Similarly, setting h to smaller values can be sufficient, in practice, for *pairwise circular sequence alignment*. To test this ability of MW-MaxShiftM, pairs of $1,000$ bp-long sequences were generated randomly with varying degrees of dissimilarity—10, 20, 30 and 40%—for the Hamming distance model. Consider a single pair of two sequences x and y. A script was written to take one of the two sequences, x, cut it at a random position and swap around the two resulting segments. This essentially rotates x at an arbitrary position creating a new sequence x'. The two sequences, x' and y, would then be compared using the aforementioned method; that is, MaxShiftM was applied with $x'x'$, y, and different fixed lengths h, ranging from 16 to 512.

When MaxShiftM found the cell with the lowest number of errors, the coordinates (i, j) returned by the program could be used to rotate sequence x' back to (ideally) its original state, thereby creating a new sequence x''. To evaluate MaxShiftM for this task, the newly created x'' sequence was compared against y using the standard dynamic programming algorithm. Ideally, MaxShift would find that this distance is equal or close to the one between x and y. The results of comparing x'' and y, for varying degrees of dissimilarity, using different fixed lengths h are shown in Table I. The *control* column denotes the original number of errors between the randomly generated sequences prior the random rotation of x takes place.

	Fixed length h						
control	**16**	**32**	**64**	**128**	**256**	**512**	**1,000**
100	100	100	100	100	100	100	100
200	200	200	200	200	200	200	200
300	300	300	300	300	300	300	300
400	400	400	400	400	400	400	400

Table I: Hamming distance between x'' and y for different fixed lengths h

As can be observed from Table I, the results have 100% match with the control column denoting that all sequences were rotated back to their original state. This in turn implies that algorithm MaxShiftM can be used for the application of pairwise circular sequence alignment under the Hamming distance model.

EXPERIMENT 3: APPLICATION II (REAL DATA)

We have incorporated our implementation in BEAR [2], a tool for improving multiple circular sequence alignment, as follows. Given a set of molecular sequences $x_0, x_1, \ldots, x_{d-1}$ as input, the objective of BEAR is to compute an array R of size d, such that $\mathsf{R}[j]$, for all $0 \leq j < d$, stores a sufficiently good rotation of x_j. Then $x_0^{\mathsf{R}[0]}, x_1^{\mathsf{R}[1]}, \ldots, x_{d-1}^{\mathsf{R}[d-1]}$ could be used as the input dataset for a conventional multiple sequence alignment algorithm to obtain the alignment. For clarity of presentation, we assume that $m = |x_0| = |x_1| = \ldots = |x_{d-1}|$.

In order to compute array R, the idea in BEAR is to apply MaxShiftM for fixed-length approximate string matching, for every pair of strings (x_i, x_j), with $x'_i = x_i[0\,..\,m-1]x_i[0\,..\,h-1]$ and $x'_j = x_j[0\,..\,m-1]$, for some $1 \leq h \leq m$. Notice that cell $\mathsf{D}'[p,q]$, say $\mathsf{D}'[p,q] = e$, denotes that factor $x'_i[p-h\,..\,p-1]$ matches factor $x'_j[q-h\,..\,q-1]$ with e mismatches. Equivalently, factor $x_i^{p \bmod m}[m-h\,..\,m-1]$ matches factor $x_j^{q \bmod m}[m-h\,..\,m-1]$ with e mismatches. Hence setting $h := m$ solves *exactly* the approximate circular string matching problem; however setting h to smaller values can be sufficient in practice for the considered application. Given the output of this approach, for all pairs of strings, BEAR constructs a matrix M of size $d \times d$ of pairs, such that $\mathsf{M}[i,j]$ stores (e,r), denoting that the Hamming distance between some factor of length h of $x_i^{\mathsf{M}[i,j].r}$ and some factor of x_j is $\mathsf{M}[i,j].e$. This step requires time $\mathcal{O}(d^2m^2\lceil h/w \rceil)$, which, in practice, is much faster than the $\mathcal{O}(d^2m^3)$-time algorithm implemented in cyclope [15]. After constructing matrix M, BEAR applies standard agglomerative hierarchical clustering to obtain array R.

To test the efficiency and accuracy of our methodology, we compared BEAR to cyclope using real data. As input datasets we used three sets of MtDNA sequences: the first set includes sequences of 16 primates; the second set includes sequences of 12 mammals; and the last one is a set of 19 distantly-related sequences (the 16 primates, plus the *Drosophila melanogaster*, the *Gallus gallus*, and the *Crocodylus niloticus*). The MtDNA genome size for each sequence in the datasets is between 16 and 20 Kbp. To ensure a fair efficiency comparison between the two programmes, we made sure that they both produce a unique phylogenetic tree (using ClustalW [14] for multiple sequence alignment and RAxML [23] for phylogenetic reconstruction) by computing the pairwise Robinson and Foulds (RF) distance [21] of the inferred trees. The results in Table II using a single core show that BEAR can accelerate the computations by more than a factor of 20 compared to cyclope, producing, via ClustalW and RAxML, *identical* trees.

Dataset	BEAR	cyclope	RF distance
First set (Primates)	2m11s	41m46s	0
Second set (Mammals)	1m15s	26m19s	0
Third set (Primates et al)	3m01s	61m35s	0

Table II: Execution-time comparison and pairwise RF distance using real data

VI. CONCLUSION

Approximate string matching is a core computational task in molecular biology and has been extensively studied over the past decades. Fixed-length approximate string matching is a generalisation of this problem that has beneficial applications in biology and beyond. MaxShiftM is a bit-parallel algorithm for fixed-length approximate string matching under the Hamming distance model. The computation in this algorithm is realised by dynamic programming techniques on bit-vectors. A straightforward implementation of MaxShift imposes a restriction on the maximal length of any represented alignment: it must be less than or equal to the size of the computer word.

In this article, we provided the full details of a generalised implementation of the word-level operations used for the maintenance of the dynamic programming matrix. In particular, we showed how MW-MaxShiftM can support arbitrarily long alignments that exceed the size of the computer word and successfully carry out its functionality in time $\mathcal{O}(m\lceil h/w \rceil n)$. Moreover, we showed some preliminary results on how our implementation can be used to improve the efficiency and accuracy of multiple circular sequence alignment in terms of the inferred likelihood-based phylogenies.

The main advantage of using MW-MaxShiftM over a naïve implementation of MaxShift would be the ability to use arbitrarily large fixed lengths to measure similarity between sequences. This would in turn provide a more confident similarity measure, especially if the sequences had many repeating factors; larger fixed lengths of factors would more likely find distinct regions in the sequences under study.

Future work will concentrate on: reducing the memory footprint and running time through more technical low-level optimisations; and extending our approach to the edit distance model.

ACKNOWLEDGMENT

SPP is supported by a Research Grant (#RG130720) awarded by the Royal Society. AR is supported by the Graduate Teaching Assistant scheme of the Department of Informatics at King's College London.

REFERENCES

[1] Ricardo A. Baeza-Yates and Gonzalo Navarro. A faster algorithm for approximate string matching. In Daniel S. Hirschberg and Eugene W. Myers, editors, *Proceedings of the seventh annual Symposium on Combinatorial Pattern Matching (CPM 1996)*, volume 1075 of *Lecture Notes in Computer Science*, pages 1–23, UK, 1996. Springer.

[2] Carl Barton, Costas S. Iliopoulos, Ritu Kundu, Solon P. Pissis, Ahmad Retha, and Fatima Vayiani. Accurate and efficient methods to improve multiple circular sequence alignment. (submitted).

[3] Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Fast algorithms for approximate circular string matching. *Algorithms for Molecular Biology*, 9(9), 2014.

[4] Carl Barton, Costas S. Iliopoulos, and Solon P. Pissis. Average-case optimal approximate circular string matching. In A.-H. Dediu, E. Formenti, C. Martn-Vide, and B Truthe, editors, *Language and Automata Theory and Applications*, volume 8977 of *Lecture Notes in Computer Science*, pages 85–96. Springer Berlin Heidelberg, 2015.

[5] Maxime Crochemore, Costas S. Iliopoulos, and Yoan J. Pinzon. Speeding-up Hirschberg and Hunt-Szymanski LCS algorithms. *Fundamenta Informaticae*, 56(1–2):89–103, 2002.

[6] Maxime Crochemore, Costas S. Iliopoulos, and Solon P. Pissis. A parallel algorithm for fixed-length approximate string-matching with k-mismatches. In Tapio Elomaa, Heikki Mannila, and Pekka Orponen, editors, *Algorithms and Applications*, volume 6060 of *Lecture Notes in Computer Science*, pages 92–101. Springer Berlin Heidelberg, 2010.

[7] Guido Fritzsch, Martin Schlegel, and Peter F. Stadler. Alignments of mitochondrial genome arrangements: Applications to metazoan phylogeny. *Journal of Theoretical Biology*, 240(4):511–520, 2006.

[8] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4(1):33–72, 1988.

[9] Zvi Galil and Kunsoo Park. An improved algorithm for approximate string matching. *SIAM Journal of Computing*, 19(6):989–999, 1990.

[10] Ana Goios, Lusa Pereira, Molly Bogue, Vincent Macaulay, and Antnio Amorim. mtDNA phylogeny and evolution of laboratory mouse strains. *Genome Research*, 17(3):293–298, 2007.

[11] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162(3):705–708, December 1982.

[12] Costas Iliopoulos, Laurent Mouchard, and Yoan Pinzon. The Max-Shift algorithm for approximate string matching. In Gerth Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela, editors, *Proceedings of the fifth International Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 13–25, Denmark, 2001. Springer.

[13] Augustine Kong, Michael L. Frigge, Gisli Masson, Soren Besenbacher, Patrick Sulem, Gisli Magnusson, Sigurjon A. Gudjonsson, Asgeir Sigurdsson, Aslaug Jonasdottir, Adalbjorg Jonasdottir, Wendy S. Wong, Gunnar Sigurdsson, G. Bragi Walters, Stacy Steinberg, Hannes Helgason, Gudmar Thorleifsson, Daniel F. Gudbjartsson, Agnar Helgason, Olafur Th T. Magnusson, Unnur Thorsteinsdottir, and Kari Stefansson. Rate of de novo mutations and the importance of father's age to disease risk. *Nature*, 488(7412):471–475, August 2012.

[14] M.A. Larkin, G. Blackshields, N.P. Brown, R. Chenna, P.A. McGettigan, H. McWilliam, F. Valentin, I.M. Wallace, A. Wilm, R. Lopez, J.D. Thompson, T.J. Gibson, and D.G. Higgins. Clustal W and Clustal X version 2.0. *Bioinformatics*, 23(21):2947–2948, 2007.

[15] Axel Mosig, Ivo L. Hofacker, and Peter F. Stadler. Comparative Analysis of Cyclic Sequences: Viroids and other Small Circular RNAs. In Daniel H. Huson, Oliver Kohlbacher, Andrei N. Lupas, Kay Nieselt, and Andreas Zell, editors, *German Conference on Bioinformatics*, volume 83 of *LNI*, pages 93–102. GI, 2006.

[16] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

[17] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[18] Solon P. Pissis. MoTeX-II: structured MoTif eXtraction from large-scale datasets. *BMC Bioinformatics*, 15:235, 2014.

[19] Solon P. Pissis, Alexandros Stamatakis, and Pavlos Pavlidis. MoTeX: A word-based HPC tool for MoTif eXtraction. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, BCB'13, pages 13–22, New York, NY, USA, 2013. ACM.

[20] Peter Rice, Ian Longden, and Alan Bleasby. EMBOSS: The European Molecular Biology Open Software Suite. *Trends in Genetics*, 16(6):276–277, 2000.

[21] D.F. Robinson and L.R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1-2):131 – 147, 1981.

[22] Peter H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[23] Alexandros Stamatakis. RAxML version 8: A tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 2014.

[24] Michael Stratton. Genome resequencing and genetic variation. *Nat Biotech*, 26(1):65–66, 2008.

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

[26] Esko Ukkonen and Derick Wood. Approximate string matching with suffix automata. *Algorithmica*, 10(5):353–364, 1993.

[27] Zhang Wang and Martin Wu. Phylogenomic reconstruction indicates mitochondrial ancestor was an energy parasite. *PLoS ONE*, 10(9):e110685, 2014.

[28] Alden H. Wright. Approximate string matching using within-word parallelism. *Software - Practice and Experience*, 24(4):337–362, 1994.

[29] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.