BIOINFORMATICS ORIGINAL PAPER Vol. 24 no. 6 2008, pages 791–797
doi:10.1093/bioinformatics/btn032

Sequence analysis

Compressed indexing and local alignment of DNA

T. W. Lam[1,*], W. K. Sung[2], S. L. Tam[1], C. K. Wong[1] and S. M. Yiu[1]

[1]Department of Computer Science, University of Hong Kong, Hong Kong, China and [2]Department of Computer Science, National University of Singapore, Singapore

Received on August 29, 2007; revised on December 8, 2007; accepted on January 22, 2008

Advance Access publication January 28, 2008

Associate Editor: Thomas Lengauer

ABSTRACT

Motivation: Recent experimental studies on compressed indexes (BWT, CSA, FM-index) have confirmed their practicality for indexing very long strings such as the human genome in the main memory. For example, a BWT index for the human genome (with about 3 billion characters) occupies just around 1 G bytes. However, these indexes are designed for exact pattern matching, which is too stringent for biological applications. The demand is often on finding local alignments (pairs of similar substrings with gaps allowed). Without indexing, one can use dynamic programming to find all the local alignments between a text T and a pattern P in $O(|T||P|)$ time, but this would be too slow when the text is of genome scale (e.g. aligning a gene with the human genome would take tens to hundreds of hours). In practice, biologists use heuristic-based software such as BLAST, which is very efficient but does not guarantee to find all local alignments.

Results: In this article, we show how to build a software called BWT-SW that exploits a BWT index of a text T to speed up the dynamic programming for finding all local alignments. Experiments reveal that BWT-SW is very efficient (e.g. aligning a pattern of length 3 000 with the human genome takes less than a minute). We have also analyzed BWT-SW mathematically for a simpler similarity model (with gaps disallowed), and we show that the expected running time is $O(|T|^{0.628}|P|)$ for random strings. As far as we know, BWT-SW is the first practical tool that can find all local alignments. Yet BWT-SW is not meant to be a replacement of BLAST, as BLAST is still several times faster than BWT-SW for long patterns and BLAST is indeed accurate enough in most cases (we have used BWT-SW to check against the accuracy of BLAST and found that only rarely BLAST would miss some significant alignments).

Availability: www.cs.hku.hk/~ckwong3/bwtsw

Contact: twlam@cs.hku.hk

1 INTRODUCTION

The decoding of different genomes, in particular the human genome, has triggered a lot of bioinformatics research. In many cases, it is required to search the human genome (called the text below) for different patterns (say, a gene of another species). Exact matching is usually unlikely and may not make sense. Instead, one wants to find local alignments, which are pairs of similar substrings in the text and pattern, possibly with gaps (see, e.g. Gusfield, 1997). Typical biological applications require a minimum similarity of 75% (match 1 point; mismatch −3 points) and a minimum length of 18–30 characters (see Section 2 for details).

To find all local alignments, one can use the dynamic programming algorithm by Smith and Waterman (1981), which uses $O(nm)$ time, where n and m are the length of the text and pattern, respectively. This algorithm is, however, too slow for a large text like the human genome. Our experiment shows that it takes more than 15 h to align a pattern of 1000 characters against the human genome. In real applications, patterns can be genes or even chromosomes, ranging from a few thousand to a few hundred million characters, and the SW algorithm would require days to weeks. As far as we know, there does not exist any practical solution for finding all local alignments in this scale. At present, BLAST (Altschul *et al.*, 1990, 1997) a heuristic method, is widely used to find local alignments. BLAST is very efficient (e.g. it takes 10–20 s to, align a pattern of 1000 characters against the human genome). Yet, BLAST does not guarantee to find all local alignments. The past few decade has witnessed different techniques including indexing to improve the heuristic and speed of BLAST (e.g. Burkhardt *et al.*, 1999; Cao *et al.*, 2005; Giladi *et al.*, 2002; Li *et al.*, 2004; Ozturk and Ferhatosmanoglu, 2003). This article, however, revisits the problem of finding all local alignments. We attempt to speed up the dynamic programming approach by exploiting the recent breakthrough on text indexing.

1.1 Indexing and dynamic programming

Let T be a text of n characters and let P be a pattern of m characters. A naive approach to finding all of their local alignments is to examine all substrings with cm characters, where c is a constant depending on the similarity model, and to align them one by one with P. Obviously, we want to avoid aligning P with the same substring at different positions of the text. A natural way is to build a suffix tree (McCreight, 1976) (or a suffix trie) of the text. Then, distinct substrings of T are represented by different paths from the root of the suffix tree. We align P against each path from the root up to cm characters using dynamic programming. The common prefix structure of the paths also gives a way to share the common parts of the dynamic programming on different paths. Specifically, we perform a pre-order traversal of the suffix tree; at each node, we maintain a dynamic programming table (DP table) for aligning the pattern and the path up to the node. We add more rows to the table as we

*To whom correspondence should be addressed.

go down the suffix tree, and delete the corresponding rows when going up the tree. Note that filling a row of the table costs $O(m)$ time. For very short patterns, the above approach performs dynamic programming on a few layers of nodes only and could be very efficient; yet there are a few issues to be resolved for this approach to be successful in general.

- **Index size:** The best known implementation of a suffix tree requires $17.25n$ bytes for a text of length n (Kurtz, 1999). For human genome, this is translated to 50 G bytes of memory, which far exceeds the 4 G capacity of a standard PC nowadays. An alternative is a hard-disk-based suffix tree, which would increase the access time several order of magnitude. In fact, even the construction of a suffix tree on a hard disk is already very time-consuming; for the human genome, it would need a week (Hunt *et al*., 2002).
- **Running time and pruning effectiveness:** The above approach requires traversing each path of the suffix tree starting from the root up to $O(m)$ characters, and computing possible local alignments starting from the first character of the path. For long patterns, this may mean visiting many nodes of the tree, using $O(nm)$ or more time. Nevertheless, we can show that at an intermediate node u, if the DP table indicates that no substring of the pattern has a positive similarity score when aligned with the path to u, then it is useless to further extend the path and we can prune the subtree rooted at u. The question is, in practice, how effective such a simple pruning strategy could be.
- **DP table:** Recall that we have to maintain a dynamic programming table at each node, which is of size $m \times d$ where d is the length of the substring represented by the node. The worst-case memory requirement is $O(m^2)$. For long patterns, say, even a gene with tens of thousands characters, the table would demand several gigabytes or more and cannot fit into the main memory.

Meek *et al*. [2003] have attempted to use a suffix tree in the hard disk to speed up the dynamic programming for finding all local alignments. As expected, success is limited to a small scale; their experiments are based on a text of length 40 M and relatively short patterns with at most 65 characters. To alleviate the memory requirement of suffix trees, we exploit the recent breakthrough on compressed indexing, which reduces the space complexity from $O(n)$ bytes to $O(n)$ bits, while preserving similar searching time. FM-index (Ferragina and Manzini, 2000, 2001), CSA (Compressed Suffix Array) (Grossi and Vitter, 2000; Sadakane, 2003), BWT (Burrow and Wheeler, 1994) are among the best known examples. In fact, empirical studies have confirmed their practicality for indexing long biological sequences to support very efficient exact matching on a PC (e.g. Hon *et al*., 2004; Lippert, 2005). For DNA sequences, BWT was found to be the most efficient and the memory requirement can be as small as $0.25n$ bytes. For the human genome, this requires only 1 G memory and the whole index can reside in the main memory of a PC. Moreover, the construction time of a BWT index is shorter, our experiment shows that it takes about an hour for the human genome.

1.2 Summary of results

Based on a BWT index in the main memory, we have built a software called BWT-SW to find all local alignments using dynamic programming. This article is devoted to the details of BWT-SW. Among others, we will present how to use a BWT index to emulate a suffix trie of the text (i.e. the tree structure of all the suffices of the text), how to modify the dynamic programming to allow pruning but without jeopardizing the completeness, and how to manage the DP tables.

BWT-SW performs well in practice, even for long patterns. The pruning strategy is effective and terminates most of the paths at a very early stage. We have tested BWT-SW extensively with the human genome and random patterns of length from 500 to a 100 million. On average, a pattern of 500 characters [resp. 5000 and 1 M characters] requires at most 10 s [resp. 1 m and 2.5 h] (see Section 4.1 for more results). When compared with the Smith–Waterman algorithm, BWT-SW is at least 1000 times faster. As far as we know, BWT-SW is the first software that can find all local alignments efficiently in such a scale. We have also tested BWT-SW using different texts and patterns (see the second table in Section 4.1). In a rough sense, the timing figures of our experiments suggest that the time complexity could be in the order of $n^{0.628}m$. In Section 4, we will also present the experimental findings on the memory utilization due to the DP tables.

To better understand the efficiency of BWT-SW, we have also analyzed BWT-SW mathematically with respect to the pure match/mismatch model (i.e. gaps are not allowed) and random strings. We prove that for DNA alignment, the total number of entries filled in all the DP tables can be upper bounded by $O(n^{0.628}\, m)$, and hence BWT-SW takes $O(n^{0.628}\, m)$ time. It is probably a coincidence that the dependency on n is found to match the above experimental result; note that the experimental result and the mathematical analysis are dealing with different alignment models. It is also worth-mentioning that our analysis implies that the DP table is very sparse; in particular, when we extend a path, the number of positive entries also decreases exponentially and is eventually bounded by a constant. Thus, we can save a lot of space by storing only the entries with positive scores.

It is worth-mentioning that BWT-SW is not meant to be a replacement of BLAST; BLAST is still several times faster than BWT-SW for long patterns and BLAST is accurate enough in most cases. Using BWT-SW, we found that BLAST may miss some significant alignments (with high similarity) that could be critical for biological research, but this occurs only rarely. Specifically, we have conducted an experiment to align 8000 queries (selected from the NCBI database of the chimpanzee, mouse, chicken and zebrafish mRNA) with the human genome. These queries are of length ranging from 170 to 19 000. If all practically insignificant alignments (with E-value less than 1×10^{-10}) are ignored, BLAST only missed 0.06% of the alignments found by BWT-SW. When we look into individual species, we further observe that the missing percentage is dependent on the evolutional distance between the species and human. More details can be found in Section 5.

2 DEFINITIONS AND BASIC CONCEPTS

In this section, we give the definitions of local alignments and BWT.

2.1 Local alignments with affine gap penalty

Let x and y be two strings. A space is a special character not found in these strings.

- An alignment A of x and y maps x and y respectively to another two strings x' and y' that may contain spaces such that (i) $|x'|=|y'|$ and (ii) removing spaces from x' and y' should get back x and y, respectively; and (iii) for any i, $x'[i]$ and $y'[i]$ cannot be both spaces.
- A gap is a maximal substring of contiguous spaces in either x' or y'.
- An alignment A is composed of three kinds of regions. (i) Matched pair: $x'[i]=y'[i]$; (ii) Mismatched pair: $x'[i]\neq y'[i]$ and both are not spaces; (iii) Gap: either $x'[i..j]$ or $y'[i..j]$ is a gap. Only a matched pair has a positive score a, a mismatched pair has a negative score b and a gap of length r also has a negative score $g+rs$ where g, $s<0$. For DNA, the most common scoring scheme (e.g. used by BLAST) makes $a=1$, $b=-3$, $g=-5$ and $s=-2$.
- The score of the alignment A is the sum of the scores for all matched pairs, mismatched pairs and gaps. The alignment score of x and y is defined as the maximum score among all possible alignments of x and y.

Let T be a text of n characters and let P be a pattern of m characters. The local alignment problem can be defined as follows. For any $1\leq i\leq n$ and $1\leq j\leq m$, compute the largest possible alignment score of $T[h..i]$ and $P[k..j]$ where $h\leq i$ and $k\leq j$ (i.e. the best alignment score of any substring of T ending at position i and any substring of P ending at position j). Furthermore, for biological applications, we are only interested in those $T[h..i]$ and $P[k..j]$ if their alignment score attains a threshold H.

2.2 Suffix trie and BWT

Suffix trie: Given a text T, a suffix trie for T is a tree comprising all suffices of T such that each edge is uniquely labeled with a character, and the concatenation of the edge labels on a path from the root to a leaf corresponds to a unique suffix of T. Each leaf stores the starting location of the corresponding suffix. Note that a pre-order traversal of a suffix trie can enumerate all suffices of T. Furthermore, if we compress every maximal path of degree-one nodes of the suffix trie, then we obtain the suffix tree of T.

BWT: The Burrows–Wheeler transform (BWT) (Barrow and Wheeler, 1994) was invented as a compression technique. It was later extended to support pattern matching by Ferragina and Manzini (2000). Let T be a string of length n over an alphabet Σ. We assume that the last character of T is a unique special character \$, which is smaller than any character in Σ. The suffix array $SA[0,n-1]$ of T is an array of indexes such that $SA[i]$ stores the starting position of the i-th-lexicographically smallest suffix. The BWT of T is a permutation of T such that $BWT[i]=T[SA[i]-1]$. For example, if $T=$'acaacg\$', then $SA=$(Altschul *et al.*, 1990, 1997; Burkhardt *et al.*, 1999; Burrow and Wheeler, 1994; Cao *et al.*, 2005; Ferragina and Manzini, 2000, 2002; Giladi *et al.*, 2002), and $BWT=$'gc\$aaacc'.

Given a string X, let $SA[i]$ and $SA[j]$ be the smallest and largest suffices of T that have X as the prefix. The range $[i,j]$ is referred to as the *SA range of* X. Given the SA range $[i,j]$ of X, finding the SA range $[p,q]$ of zX, for any character z, can be done using the backward search technique (Ferragina and Manzini, 2000) as follows.

LEMMA 1. *Let X be a string and z be a character. Suppose that the SA range of X and zX is $[i..j]$ and $[p..q]$, respectively. Then $p=C(z)=Occ(z,\ i-1)=1$, and $q=C(z)=Occ(z,j)$, where $C(z)$ is the total number of characters in T that are lexicographically smaller than z and $Occ(z,i)$ is the total number of z's in $BWT[0..i]$.*

We can pre-compute $C(z)$ for all characters z and retrieve any entry in constant time. Using the auxiliary data structure introduced by Ferragina and Manzini (2000), computing $Occ(z,i)$ also takes constant time. Then $[p,q]$ can be calculated from $[i,j]$ in constant time. As a remark, BWT can be constructed in a more efficient way than other indexes like suffix trees. We have implemented the construction algorithm of BWT described in Hon *et al.*, (2007) it takes 50 min to construct the BWT of the human genome using a Pentinum D 3.6 GHz PC.

3 METHODS: INDEXING, DP AND PRUNING

We solve the local alignment problem in two phases.

- **Phase I.** For all $1\leq i\leq n$, $1\leq j\leq m$, and $1\leq h\leq i$, compute $A[h,i,j]$ which equals the largest alignment score of $T[h..i]$ and any substring of P ending at position j.
- **Phase II.** For all $1\leq i\leq n$, $1\leq j\leq m$, return the largest among all alignment scores $A[h,i,j]$ for different h.

In Phase I, each combination of (h,i) defines a substring of T. Thus, Phase I can be rephrased as follows: for every substring X of T and for all $1\leq j\leq m$, find the best alignment score of X and any substring of P ending at position j. In the following, we will show how to use a suffix trie of T to speed up this step. With a suffix trie, we can avoid aligning substrings of T that are identical. That is, we exploit the common prefix structure of a trie to avoid identical substrings to be aligned more than once. We use a pre-order traversal of the suffix trie to generate all distinct substrings of X. Also, we only need to consider substrings of T of length at most cm where c is usually a constant bounded by 2. This is because the score of a match is usually smaller than the penalty due to a mismatch/insert/delete, and a substring of T with more than $2m$ characters have at most m matches and an alignment score less than 0.

For each node u of depth d ($d\leq cm$) in the suffix trie of T, let $X[1..d]$ be the substring represented by this node. There may be multiple occurrences of X in T and the starting positions of these occurrences, say, $p_1,p_2,\ldots,p_w$, can be found by traversing the leaves of the subtree rooted at u. For each $1\leq j\leq m$, we compute the best possible alignment score of X and any substring of P ending at position j, or equivalently, we compute $A[p_1,p_1+d-1,j]$, $A[p_2,p_2+d-1,j],\ldots,$ $A[p_w,p_w+d-1,j]$.

The rest of this section is divided into three parts: Section 3.1 shows how to make use of a BWT index to simulate a pre-order traversal of a suffix trie. Section 3.2 gives a simple dynamic programming method to compute, for each node u on a path of the suffix trie and for all $1\leq j\leq m$, the best alignment score of the substring represented by u and

any substring of P ending at j. Section 3.3 shows that the dynamic programming on a path can be terminated as soon as we realize that no 'meaningful' alignment can be produced.

3.1 Simulating suffix trie traversal using BWT

We can make use of the backward search technique on BWT to simulate the pre-order traversal of a suffix trie to enumerate the substrings. Based on Lemma 1, we have the following corollary.

COROLLARY 2. *Given the SA range $[i,j]$ of X in T, if the SA range $[p,q]$ of zX for a character z computed by Lemma* 1 *is invalid, i.e. $p>q$, then zX does not exist in T.*

Since we use backward search, instead of constructing the BWT for T, we construct the BWT for the reversal of T. Consider a node u in the suffix trie of T, which represents the substring X. To check if u has an outgoing edge labeled with a character z, we can check if zX^{-1} exists in T^{-1}.

We can simulate the traversal of a suffix trie and enumerate the substrings represented by the nodes in the trie as follows. Assume that we are at node u in the suffix trie of T that represents the substring X, and we have already found the SA range for X^{-1} in T^{-1} using the BWT. Based on the above corollary, we can check the existence of an edge with label z from u in $O(1)$ time by computing the SA range for zX^{-1} using the BWT of T^{-1}. Then, we enumerate the corresponding substring if the edge does exist and repeat the same procedure to traverse the tree.

3.2 Dynamic programming

Consider a path from the root of the suffix trie. Subsequently we present a dynamic programming method to compute, for each node u on this path and for all $1 \le j \le m$, the best possible alignment score of the substring $X[1..d]$ represented by u and any substring of P ending at j.

For any $i \le d$ and $j \le m$, let $M^u(i,j)$ be the best alignment score of $X[1..i]$ and any substring of P ending at position j. Let $M_1^u(i,j)$, $M_2^u(i,j)$ and $M_3^u(i,j)$ be the best possible alignment score of $X[1..i]$ and a substring of P ending at position j with $X[i]$ aligned with $P[j]$, $X[i]$ aligned with a space and $P[j]$ aligning with a space, respectively. The values of $M^u(d,j)$ shows the best alignment score of $X[1..d]$ and a substring of P ending at position j.

Initial conditions:

$$\begin{aligned} M^u(0,j) &= 0 && \text{for } 0 \le j \le m. \\ M_1^u(i,0) &= -(g+is) && \text{for } 1 \le i \le d. \\ M_2^u(0,j) &= -\infty && \text{for } 0 \le j \le m. \\ M_3^u(i,0) &= -\infty && \text{for } 1 \le i \le d. \end{aligned}$$

Recurrences (for $i>1, j>1$):

$$\begin{aligned} M_1^u(i,j) &= M^u(i-1,j-1) + \delta(X[i],P[j]). \\ M_2^u(i,j) &= \max\{M_2^u(i-1,j)-s, M^u(i-1,j)-(g+s)\}. \\ M_3^u(i,j) &= \max\{M_3^u(i,j-1)-s, M^u(i,j-1)-(g+s)\}. \\ M^u(i,j) &= \max\{M_1^u(i,j), M_2^u(i,j), M_3^u(i,j)\} \end{aligned}$$

where $\delta(X[i],P[j])=a$ if $X[i]=P[j]$, otherwise $\delta(X[i],P[j])=b$. (See Section 2 for definitions of a and b.)

Consider a child v of u. Denote the substring represented by v as $X[1..d]c$. Note that when we extend the dynamic programming from node u to node v, we only need to compute a new row at each dynamic programming table of u (e.g. $M^u(d+1,j), M_1^u(d+1,j), M_2^u(d+1,j), M_3^u(d+1,j)$ for all $1 \le j \le m$. If a traversal of the suffix trie would move from node u to its parent, we erase the last row of every dynamic programming table computed at u.

3.3 Modified dynamic programming and pruning

In this section, we show how to modify the dynamic programming to enable an effective pruning. We first define what a meaningless alignment is.

3.3.1 Meaningless alignment Let A be an alignment of a substring $X=T[h..i]$ of T and a substring $Y=P[k..j]$ of P. If A aligns a prefix $X'=T[h..h']$ of X with a prefix $Y'=P[k..k']$ of Y such that the alignment score of X' and Y' is less than or equal to zero, A is said to be a *meaningless alignment*. Otherwise, A is said to be *meaningful*.

LEMMA 3. *Suppose that A is a meaningless alignment of a substring $X=T[h..i]$ and a substring $Y=P[k..j]$ with a positive score C. Then there exists a meaningful alignment for some proper suffix $X'=T[s..i]$ of X and some proper suffix $Y'=P[t..j]$ of Y with score at least C, where $h<s\le i$ and $k<t\le j$.*

PROOF. *Suppose that A aligns a prefix $X'=T[h..h']$ of X with a prefix $Y'=P[k..k']$ of Y such that the induced alignment score C' of X' and Y' is less than or equal to zero. Let $T[s]$ be the first character after $T[h']$ such that it aligns with a character of $P[k'=1..j]$ in A (denote this character as $P[t]$). Let D be the alignment score of $T[s..i]$ and $P[t..j]$ and A' be a corresponding alignment. Then, $D+C' \ge C$. Since $C' \le 0$, $D \ge C$.*
If the alignment A' is meaningless, we repeat the same argument on $T[s..i]$ and $P[t..j]$ until we obtain a suffix of X and a suffix of Y with a meaningful alignment of score $\ge C$. Since $C>0$, the lemma follows. ■

Subsequently, we show how to modify the dynamic programming to only compute the best possible score of meaningful alignments (*meaningful alignment score*). Notice that for any two strings, the best meaningful alignment score may not be the best alignment score. Nevertheless, we will show that the meaningful alignment scores are already sufficient for Phase II to report the correct answers.

3.3.2 DP for meaningful alignment score In the dynamic programming tables, entries with values less than or equal to zero will never be used.

Let u be a node in the suffix trie for T and $X[1..d]$ be the string represented by u. Let $N^u(i,j)$ be the best possible score of a *meaningful* alignment between $X[1..i]$ and a suffix of $P[1..j]$. Furthermore, $N_1^u(i,j)$, $N_2^u(i,j)$ and $N_3^u(i,j)$ are defined in a similar way as $M_1^u(i,j)$, $M_2^u(i,j)$ and $M_3^u(i,j)$. The recurrence equations are modified as follows. For any $i,j>1$,

$$N_1^u(i,j) = \begin{cases} N^u(i-1,j-1) + \delta(X[i],P[j]) & \text{if } N^u(i-1,j-1) > 0 \\ -\infty & \text{otherwise} \end{cases}$$

$$N_2^u(i,j) = \begin{cases} \max\{N_2^u(i-1,j)-s, N^u(i-1,j)-(g+s)\} & \text{if } N_2^u(i-1,j)>0 \text{ and } N^u(i-1,j)>0 \\ N_2^u(i-1,j)-s & \text{if only } N_2^u(i-1,j)>0 \\ N^u(i-1,j)-(g+s) & \text{if only } N^u(i-1,j)>0 \\ -\infty & \text{otherwise} \end{cases}$$

$$N_3^u(i,j) = \begin{cases} \max\{N_3^u(i,j-1)-s, N^u(i,j-1)-(g+s)\} & \text{if } N_3^u(i,j-1)>0, \text{ and } N^u(i,j-1)>0 \\ N_3^u(i,j-1)-s & \text{if only } N_3^u(i,j-1)>0 \\ N^u(i,j-1)-(g+s) & \text{if only } N^u(i,j-1)>0 \\ -\infty & \text{otherwise} \end{cases}$$

$$N^u(i,j) = \max\{N_1^u(i,j), N_2^u(i,j), N_3^u(i,j)\}$$

Next, we show that the scores computed by the modified dynamic programming are sufficient for Phase II to compute the correct answers, thus solving the local alignment problem.

LEMMA 4. *Let u be a node in the suffix trie for T and let $X[1..d]$ be the string represented by u. If $M^u(d,j)=C \ge H$ where H is the*

score threshold, then, there exists h in [1,d] such that $N^v(d-h=1,j)=C$ *where v is the node in the suffix trie representing the string X[h..d].*

PROOF. *If there exists a meaningful alignment for X[1..d] and P[k..j] with score = C, then h = 1 and v = u. Otherwise, based on Lemma* 3, *there exists* h *with* $1<h\leq d$ *such that there is a meaningful alignment for X[h..d] and P[k..j] with score at least C. Since* $M^u(d,j)$ *is the best possible score for X[1..d] and any substring of P ending at j,* $N^v(d-h=1,j)=C$ *where v is the node representing X[h..d].* ■

COROLLARY 5. *For any i, j, let C be the largest possible score between a substring of T ending at i and a substring of P ending at j (i.e., C is the answer for Phase II). Then there exists a node v representing a substring* $X=T[s..i]$ $(s\leq i)$ *of T such that* $N^v(i-s=1,j)=C$.

3.3.3 Pruning Strategy Since we only consider meaningful alignments, for each node in the suffix trie, when filling the dynamic programming tables, we ignore all entries with values less than or equal to zero. For a node u, if there is a row with all entries in all dynamic programming tables with values less than or equal to zero, we can stop filling the tables since all the rows below will only contain entries with values less than or equal to zero. Moreover, based on the same argument, we can prune the whole subtree rooted at u.

4 EXPERIMENTAL RESULTS AND MATHEMATICAL ANALYSIS

4.1 Efficiency of BWT-SW in practice

To see how fast BWT-SW is, we have constructed the BWT index for the human genome (NCBI Build 35) and used BWT-SW to align patterns of length from 100 to 100 M with the human genome. The query patterns are randomly selected from the mouse genome except for the query of length 100 M which is the whole mouse chromosome 15. For queries of length 10 K or shorter, we have repeated the same experiment a hundred times to get the average time. For longer patterns, we have repeated the experiments a few dozen times. Note that DNA is composed of double strands (i.e. two complimentary sequences of the same length). Instead of aligning a pattern with both strands, we align the pattern and then its reverse complement with one strand. The time reported below is the total time for aligning the pattern and its reverse complement. To avoid meaningless alignment, regions of the query that are expected to contain very little information (called *low complexity regions*; e.g. a long sequence of 'A') are masked by a standard software tool, DUST, before the alignment process. This is also the default setting of existing software such as BLAST.

The following two tables show the average time required by BWT-SW, the DP algorithm by Smith and Waterman, and BLAST. The experiments are conducted on a Pentinum D 3.0 GHz PC with 4G memory.

Query length	100	200	500	1 K	2 K
BWT-SW average time (s)	1.91	4.02	9.89	18.86	35.93
Smith–Waterman average time (K)	5.1	10.0	23.9	45.1	97.8
BLAST average time	9.7	12.58	12.52	15.23	15.82

Query length	5 K	10 K	100 K	1 M	10 M	100 M
BWT-SWaverage time (s)	82	161	1.4 K	8.9 K	34.4 K	218.2 K
BLAST	19.9	29.6	93.4	775	6.7 K	92.2 K

For patterns with thousands of characters (which is common in biological research), BWT-SW takes about 1–2 min, which is very reasonable. For extremely long patterns, say, a chromosome of 100 M, it takes about 2.5 days. In the past, finding a complete set of local alignments for such long patterns is not feasible.

To investigate how the searching time depends on the text size, we fix the pattern length and conduct experiments using different texts (chromosomes) of length ranging from 100 M to 3 G. We have repeated the study for four different query lengths. The following table shows the results.

Patern length	Text size				
	114 M	307 M	1.04 G	2.04 G	3.08 G
500	1.33	2.41	5.21	7.89	9.89
1 K	2.55	4.59	10.05	15.14	18.86
5 K	10.74	19.53	42.20	65.57	81.60
10 K	21.01	38.20	83.96	128.97	161.04

Using the above figures, we roughly estimate that the time complexity of BWT-SW is in the order of $n^{0.628}m$.[1] However, our experiments are limited, and such estimation is not conclusive. It only provides a rough explanation why BWT-SW is a 1000 times faster than the Smith–Waterman algorithm when aligning the human genome.

4.2 Mathematical analysis

To understand the performance of BWT-SW better, we have studied and analyzed a simplified model in which an alignment cannot insert spaces or gaps, and the scoring function is simply a weighted sum of the number of matched and mismatched pairs. We found that under this model, the expected total number of DP cells with positive values is upper bounded by $69n^{0.628}m$. The time required by BWT-SW is proportional to the number of DP cells to be filled, or equivalently, the number of cells with positive values. Thus, our analysis suggests that BWT-SW takes $O(n^{0.628}m)$ time under this model.[2]

We assume that strings are over an alphabet of σ characters where σ is a constant. Let x, y be strings with $d\geq 1$ characters. Suppose that x and y match in $e\leq d$ positions. Define $Score(x,y)=e-3(d-e)$ (i.e. match = 1, mismatch = −3) and

[1]For each pattern length, we fit the above data to the function $f(n)=cn^{0.628}$ where c is a fixed constant. The root-mean-square errors of the data in all four cases are within 1.21–1.63%.

[2]It is perhaps a coincidence that the dependency on n is found to match the experimental result in Section 4.1. Note that Sections 4.1 and 4.2 are based on different alignment models, one with gaps and one without.

define $f(x)$ to be the number of length-d strings y such that $Score(x, y) > 0$. Note that $f(x)$ captures the number of ways to modify x into strings y such that $Score(x, y) > 0$; thus $f(x) = f(x')$ for any length-d string x'. It is useful to overload f such that the domain of f includes integers, and $f(d)$ is defined to be $f(x)$, where $|x| = d$.

LEMMA 6. *$f(d) \leq k_1 k_2^d$, where $k_1 = (\sigma - 1/\sigma - 2)(4e/\sqrt{2\pi})$, and $k_2 = (4e\ (\sigma - 1))^{1/4}$.*

PROOF. *Let x be a string of length d. $f(d)$ is the number of strings of length-d that has at most $\lfloor d/4 \rfloor$ differences with x. Using the fact that $\binom{d}{i} \leq \left(\frac{1}{\sqrt{2i\pi}}\right)\left(\frac{de}{i}\right)^i$ (derived from Stirling's approximation), we have $\binom{d}{\lfloor d/4 \rfloor} \leq \binom{d}{\lceil d/4 \rceil} \leq \left(\frac{1}{\sqrt{2d\pi}}\right)\left(\frac{de}{\lceil d/4 \rceil}\right)^{\lceil d/4 \rceil} \leq \frac{1}{\sqrt{2\pi}}\left(\frac{de}{d/4}\right)^{d/4+1} = \frac{4e}{\sqrt{2\pi}}(4e)^{d/4}$. Then,*

$$f(d) \leq \sum_{i=0}^{\lfloor d/4 \rfloor} (\sigma - 1)^i \binom{d}{i} \leq \left(\frac{\sigma - 1}{\sigma - 2}\right)(\sigma - 1)^{d/4} \binom{d}{\lfloor d/4 \rfloor} = k_1 k_2^d,$$

where $k_1 = \frac{\sigma - 1}{\sigma - 2}\frac{4e}{\sqrt{2\pi}}$ and $k_2 = (4e\ (\sigma - 1))^{1/4}$. ■

There are σ^d strings of length d, and the following fact follows.

FACT 1. *Let x be a string of length d, for any randomly chosen length-d string y, the probability that Score $(x, y) > 0$ is $f(d)/\sigma^d$.*

Let T be a text of n characters and let R be the suffix trie of T. Let P be a pattern of m characters. For any node u in R, let $X[1..d]$ be the string represented by u. Let N^u be the dynamic programming table for u such that $N^u(d, j)$ denote $Score(X, P')$ where P' is a length-d substring of P ending at position j. In the following, we try to bound the expected total number of positive entries $N^u(d, j)$ for all u, d, j. Let $c = \lfloor \log_\sigma n \rfloor$.

LEMMA 7. *The expected total number of positive entries $N^u(d, j)$ for all nodes u at depth d is at most $mf(d)$, if $d \leq c$, and $m[n(f(d)/\sigma^d)]$, if $d > c$.*

PROOF. *For a substring P' of P ending at position j with length d, for any string $X[1..d]$, $Score(X, P') > 0$ if and only if the entry $N^u(d, j) > 0$ where u is a node in R representing X. There are $m - d + 1$ substrings of P with length d. For each of these substrings P', there are $f(d)$ strings X such that Score $(X, P') > 0$.* ■

Case 1. For any d, the total number of positive entries in $N^u(d, j)$ for all nodes u of depth d is at most $mf(d)$.

Case 2. For $d > c$, the bound for Case 1 still holds, but the expected total number of positive entries can have a tighter bound. Based on Fact 1, the expected number of these X strings appearing in T is $n(f(d)/\sigma^d)$. So, the expected total number of positive entries $N^u(d, j)$ for all nodes u at depth d is at most $m[n(f(d)/\sigma^d)]$

LEMMA 8. *For any d in $[1, c-1]$, the expected total number of positive entries $N^u(d, j)$ for all nodes u of depth d is at most $\sum_{d=1}^{c-1} mf(d) \leq c_1 mn^{c_2}$ where c_1, c_2 are constants and $c_2 < 1$.*

PROOF. $\sum_{d=1}^{c-1} mf(d) \leq k_1 m \sum_{d=-\infty}^{c-1} k_2^d = k_1 m\left(\sum_{d=1}^{\infty} k_2^{-d}\right) k_2^c = (k_1 m/(k_2 - 1))k_2^{\lfloor \log_\sigma n \rfloor} = (k_1/(k_2 - 1))\, mn^{\log_\sigma k_2} = c_1 mn^{c_2}$, where $c_1 = \frac{k_1}{k_2 - 1}$ and $c_2 = \log_\sigma k_2$. ■

LEMMA 9. *For all d, j and node u of depth d in* [c, m], *the expected total number of positive entries N^u(d, j) for all nodes at depth d is at most $\sum_{d=c}^{m}(nmf(d)/\sigma^d) \leq c'_1 mn^{c'_2}$ where c'_1, c'_2 are constants and $c'_2 < 1$.*

PROOF. Recall that $c = \lfloor \log_\sigma n \rfloor$, *i.e.* $n/\sigma^c \leq \sigma$. $\sum_{d=c}^{m} nm\frac{f(d)}{\sigma^d} \leq k_1 mn \sum_{d=c}^{\infty}\left(\frac{k_2}{\sigma}\right)^d = \frac{k_1 mn}{1 - k_2/\sigma}\left(\frac{k_2}{\sigma}\right)^c \leq \frac{k_1\sigma}{\sigma - k_2}\left(\frac{n}{\sigma^c}\right) mk_2^{\log_\sigma n} \leq \frac{k_1\sigma^2}{\sigma - k_2} mn^{\log_\sigma k_2} = c'_1 mn^{c'_2}$, where $c'_1 = \frac{k_1\sigma^2}{\sigma - k_2}$ and $c'_2 = \log_\sigma k_2$. ■

Based on Lemmas 8 and 9, we have the following corollary.

COROLLARY 10. *The expected total number of positive entries N^u(d, j) for all u, d, j is 69 $mn^{0.628}$.*

PROOF. *By lemmas 8 and 9, the expected total number of positive entires $N^u(d, j)$ is $c_1 mn^{c_2} + c'_1 mn^{c'_2}$. Substituting $\sigma = 4$ for DNA, we have approximately $k_1 = 6.5066$, $k_2 = 2.3898$, $c_1 = 4.6816$, $c'_1 = 64.6557$ and $c_2 = c'_2 = 0.6285$, and hence $69.3373mn^{0.6285}$ positive entries.* ■

4.3 Memory for DP Tables

In the DP tables of Section 3.3, only the positive entries have to be stored. We store them in a compact manner as follows. We use a big single array **B** to store entries of all rows of a DP table N. Let n_i be the number of entries, say $N(i, j_{i_1})$, $N(i, j_{i_2}), \ldots, N(i, j_{n_1})$, in the i-th row. Let $k_i = \sum_{r=1}^{i} n_r$. The entry $N(i, j_{i_\ell})$ is stored at $\mathbf{B}[k_{i-1} + \ell]$. For each entry, we store the coordinates (i, j_{i_ℓ}) and the score. We also store the starting index of **B** for each row. Moving from node u at depth y of the trie to its child v, we add a new row for v starting at $\mathbf{B}[k_y + 1]$. If we go up from node v to u, we reuse the entries in **B** from $\mathbf{B}[k_y + 1]$.

Regarding the memory required by the DP tables, it is related to the maximum number of table entries to be maintained throughout the whole searching process. This number is very small in all test cases. For example, for human genome as the text, the maximum number of entries in the DP tables are about 2600 and 22 000 for patterns of length 100 K and 1 M, respectively. The actual memory required are about 20 K and 10 M, respectively. In fact, based on the simplified model of Section 4.2, we can show that the expected number of DP table entries along any path of the suffix trie is bounded by cm where c is a constant and m is the pattern length. The memory required for DP tables is neglectable when compared to the memory for the BWT data structure: For a DNA sequence of n characters, $2n$ bits are needed for BWT entries; $2n$ bits for storing the original sequence; n bits for the auxiliary data structures. So, altogether it translates to about 2G memory for the human genome.

5 DISCUSSION: COMPLETENESS OF BLAST

From the biological point of view, a local alignment with a high similarity measure is not necessarily a very significant one as it may just occur by random. The biological community adopts a significant measure, called Expectation value (E-value), to evaluate each alignment. Roughly speaking, the E-value of an alignment reflects the expected number of alignments between the text and the pattern that have the same or even better similarity score. The lower the E-value, the more significant the

alignment is. For the detailed calculation, one can refer to Karlin and Altschul (1990). Usually, an alignment is considered to be significant in practice if its *E*-value is less than 1×10^{-10}.

It is well-known that BLAST does not guarantee to report all alignments, but it is not clear how many answers it would miss. With the BWT-SW, we have done some experiments to confirm that BLAST would miss some significant alignments but only rarely. We have conducted an experiment using a set of 8000 queries, which are constructed from 2000 mRNA from each of the four species: chimpanzee, mouse, chicken and zebrafish. These queries are of length ranging from 170 to 19 000, with an average of 2700. Following the default setting of BLAST, the low-complexity regions of the queries are masked by DUST before the alignment. The following table shows the missing percentage of BLAST when compared with BWT-SW. Each row shows the figures for the alignments with E-value less than or equal to a given value.

E-Value ≤	Percentage of missing				
	Chimpanzee	Mouse	Chicken	Zebrafish	All Four Species
10^{-16}	0.00	0.03	0.05	0.06	0.01
10^{-15}	0.00	0.03	0.05	0.06	0.02
10^{-14}	0.00	0.04	0.06	0.06	0.02
10^{-13}	0.00	0.03	0.07	0.14	0.02
10^{-12}	0.01	0.04	0.10	0.17	0.03
10^{-11}	0.02	0.05	0.11	0.28	0.05
10^{-10}	0.02	0.07	0.13	0.39	0.06
10^{-9}	0.03	0.09	0.16	0.60	0.08
10^{-8}	0.05	0.11	0.25	0.77	0.12
10^{-7}	0.10	0.19	0.31	0.81	0.18
10^{-6}	0.17	0.31	0.45	1.08	0.28
10^{-5}	0.32	0.47	0.70	1.45	0.45
10^{-4}	0.57	0.88	0.99	1.81	0.75
10^{-3}	0.99	1.36	1.25	2.25	1.17
10^{-2}	1.69	2.11	1.68	2.61	1.84
10^{-1}	2.70	2.97	2.33	2.86	2.76

If we focus on relatively significant alignments, say, with *E*-value less than or equal to 1×10^{-10}, BLAST only misses 0.06% when all 8000 queries are considered together (precisely, 49 out of 81 054 alignments found by BWT-SW are missed). We also observe that the missing percentage has a dependency on the evolutionary distance between the species and human. For example, for *E*-value less than or equal to 1×10^{-10}, the missing percentages for queries derived from chimpanzee, mouse, chicken, and zebrafish are respectively 0.02%, 0.07%, 0.13% and 0.39% (or equivalently, 10/46903, 13/19841, 15/11482 and 11/2828). In conclusion, our experiment indicates that BLAST is accurate enough in most cases, yet the few alignments missed could be critical for biological research.

ACKNOWLEDGEMENT

The project is partially supported by Hong Kong RGC Grant HKU7140/064.

Conflict of Interest: none declared.

REFERENCES

Altschul,S.F. *et al*. (1990) Basic local alignment search tool. *J. Mol. Biol*., **215**, 403–410.
Altschul,S.F. *et al*. (1997) Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucl. Acids Res*., **25**, 3389–3402.
Burkhardt,S. *et al*. (1999) q-Gram based database searching using a suffix array (quasar). *RECOMB*, 77–83.
Burrow,M. and Wheeler,D.J. (1994) A block-sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, California.
Cao,X. *et al*. (2005) Indexing DNA sequences using q-grams. *DASFAA*, 4–16.
Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. *FOCS*, 390–398.
Ferragina,P. and Manzini,G. (2001) An experimental study of an opportunistic index. *SODA*, 269–278.
Giladi,E. *et al*. (2002) SST: An algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size. *Bioinformatics*, **18**, 873–877.
Grossi,R. and Vitter,J.S. (2000) Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *STOC*, , 397–406.
Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.
Healy,J. *et al*. (2003) Annotating large genomes with exact word matches. *Genomes Research*, **13**, 2306–2315.
Hon,W.K. *et al*. (2007) Constructing compressed suffix arrays with large alphabets. *Algorithmica*, **48**, 23–36.
Hon,W.K. *et al*. (2004) Practical aspects of compressed suffix arrays and FM-Index in searching DNA sequences. *ALENEX/ANALC*, 31–38.
Hunt,E. *et al*. (2002) Database indexing for large DNA and protein sequence collections. *The VLDB J*., **11**, 256–271.
Karlin,S. and Altschul,S.F (1990) Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences*, **87**, 2264–2268.
Kurtz,S. (1999) Reducing the space requirement of suffix trees. *Software - Practice and Exp*., **29**, 1149–1171.
Li,M. *et al*. (2004) PatterHunter II: Highly sensitive and fast homology search. *J. Bioinformatics Comput. Biol*., **2**, 417–440.
Lippert,R. (2005) Space-efficient whole genome comparisons with Burrows-Wheeler transforms. *J. Comput. Biol*., **12**, 407–415.
McCreight,E.M. (1976) A space-economical suffix tree construction algorithm. *J. ACM*, **23**, 262–272.
Meek,C. *et al*. (2003) OASIS: An online and accurate technique for local-alignment searches on biological sequences. *VLDB*, 910–921.
Ozturk,O. and Ferhatosmanoglu,H. (2003) Effective indexing and filtering for similarity search in large biosequence databases. *BIBE*, 359–366.
Sadakane,K. (2003) New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, **48**, 294–313.
Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol*., **147**, 195–197.
Williams,H.E. and Zobel,J. (2002) Indexing and retrieval for genomic databases. *IEEE Trans. Knowledge Data Eng*., **14**, 63–78.