BIOINFORMATICS

Vol. 20 no. 13 2004, pages 2122–2134
doi:10.1093/bioinformatics/bth212

Speeding up whole-genome alignment by indexing frequency vectors

Tamer Kahveci, Vebjorn Ljosa and Ambuj K. Singh*

Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106-5110, USA

Received on June 21, 2003; revised on January 29, 2004; accepted on February 8, 2004
Advance Access publication April 8, 2004

ABSTRACT

Motivation: Many biological applications require the comparison of large genome strings. Current techniques suffer from high computational and I/O costs.

Results: We propose an efficient technique for local alignment of large genome strings. A space-efficient index is computed for one string, and the second string is compared with this index in order to prune substring pairs that do not contain similar regions. The remaining substring pairs are handed to a hash-table-based tool, such as BLAST, for alignment. A dynamic strategy is employed to optimize the number of disk seeks needed to access the hash table. Additionally, our technique provides the user with a coarse-grained visualization of the similarity pattern, quickly and before the actual search. The experimental results show that our technique aligns genome strings up to two orders of magnitude faster than BLAST. Our technique can be used to accelerate other search tools as well.

Availability: A web-based demo can be found at http://bioserver.cs.ucsb.edu/. Source code is available from the authors on request.

Contact: tamer@cs.ucsb.edu

INTRODUCTION

The growth in the amount of genomic information has spurred increased interest in large-scale comparison of genetic strings. We propose an efficient algorithm for aligning large genome strings. For one of the strings, we construct an index structure, called F-index, which is ∼2% of the size of database. Next, we construct a boolean match table by partitioning one of the strings into substrings and searching these substrings in the F-index of the other string. The columns of the match table correspond to substrings of the first genome, the rows to substrings of the second genome. An entry in the match table is marked as true if the corresponding substrings potentially contain similar patterns, and false otherwise. Finally, we divide the match table into either horizontal or vertical slices, and give each slice to BLAST (Altschul *et al*., 1990), or any other alignment tool, for processing. The size of the slices is such that the BLAST computation fits in main memory. We call our technique MAP, for match-table-based pruning.

Experimental results show that MAP runs up to two orders of magnitude faster than BLAST without decreasing the output quality. Furthermore, MAP can work well even with small memory sizes. This drastic reduction in CPU and I/O cost makes homology searches viable on desktop PCs. The filtering and scheduling techniques of MAP can easily be used to speed up and reduce the memory requirements of any of the current local alignment tools. MAP also provides the user with a coarse-grained visualization of the similarity pattern between the strings prior to the actual search.

RELATED WORK

The dynamic programming solution to the problem of finding the best alignment between two strings of lengths m and n runs in $O(mn)$ time and space (Needleman and Wunsch, 1970; Smith and Waterman, 1981). For large strings, this technique is infeasible in terms of both time and space. Myers (1986) improved the time and space complexity to $O(rn)$, where r is the amount of allowed error, by maintaining only the required part of the distance matrix. However, for large error rates, r is $O(m)$, so the complexity is still $O(mn)$. SIM (Huang and Miller, 1991), which uses dynamic programming to find all the alignments, is extremely slow for large strings. GLASS (Batzoglou *et al*., 2000) accelerates the dynamic programming solution by finding exactly matching long substrings first, but the time and space complexity are still high, since the extraction of k-mers is required.

Many heuristic-based search tools have been developed to align strings faster. They fall into two categories: hash-table-based tools and suffix-tree-based tools.

Some of the important hash-table-based tools are FASTA (Pearson and Lipman, 1988), BLAST, MegaBLAST (Zhang *et al*., 2000), BL2SEQ (Tatusova and Madden, 1999), WU-BLAST (Gish, 1995), SENSEI (States and Agarwal, 1996), FLASH (Califano and Rigoutsos, 1993), PipMaker (BLASTZ) (Schwartz *et al*., 2000), PatternHunter (Ma *et al*., 2002), and BLAT (Kent, 2002). These techniques are similar in spirit: they construct a hash table on one of the strings, and

*To whom correspondence should be addressed.

insert all substrings of a certain length l. The value of l varies between the tools and for different applications (e.g. BLAST uses $l = 11$ for nucleotides and $l = 3$ for proteins). The tools start by finding exactly matching substrings (known as seeds) of length l using this hash table. In a second phase, the seeds are extended in both directions, and combined, if possible, in order to find better alignments. The main difference between the tools is that they use different seed lengths and seed extension strategies. Current hash-table-based search tools handle short queries well, but become very inefficient, in terms of both time and space, for long queries.

A number of homology search tools are based on suffix trees and derivatives, see Gusfield (1997). These include MUMmer (Delcher *et al*., 1999), QUASAR (Burkhardt *et al*., 1999), REPuter (Kurtz and Schleiermacher, 1999) and AVID (Bray *et al*., 2003). QUASAR builds a suffix array on one of the strings, and counts the number of exactly matching seeds using this suffix array. If the number of seeds for a region exceeds a prespecified threshold, the region is searched using BLAST. REPuter builds a suffix tree on a string to find repetitions directly. MUMmer builds the suffix tree on both of the strings to find maximal unique matches. There are two significant problems with the suffix-tree approach: (1) Suffix trees manage mismatches inefficiently. They are good for highly similar strings, but fail to recognize more distant homologies. (2) Suffix trees have a high space overhead. The suffix tree in MUMmer, e.g. uses $37n$ bytes of memory, where n is the input length (Delcher *et al*., 1999), although with careful implementation, this can be reduced to $8n$. AVID (Bray *et al*., 2003) handles mismatches and gaps by using a variant of the Smith–Waterman algorithm once the anchors have been selected with the help of suffix trees.

CHAOS (Brudno and Morgenstern, 2002) indexes k-mers using a threaded trie, which is a cross between a suffix tree and a hash table in spirit. A threaded trie is a tree structure that stores one node for each common prefix of all the k-mers in one string. Each node also contains a pointer, called a back pointer, to the next k-mer. LAGAN (Brudno *et al*., 2003) and DIALIGN (Morgenstern, 1999; Morgenstern *et al*., 1998) employ CHAOS to find anchors for global alignment. Brudno *et al*. (2003) also use CHAOS to find *glocal* alignment, which is a combination of global and local alignments.

All the tools mentioned above require data structures larger than the database—some of them more than two orders of magnitude larger. This makes them infeasible for large-scale genome comparison. In contrast, our data structures fit in main memory: the size of the F-index is only $\sim$2% of the database size, and the match table can be adapted to whatever amount of main memory is available.

ALGORITHMS

Our technique is based on representing substrings of the two genomes by points in a multi-dimensional integer space, which we call frequency space. This transformation allows for efficient computation of an upper bound for the alignment score between the substrings represented by these points in the frequency space. If the score for two points in the frequency space is lower than a user-defined cutoff value, the actual alignment score of the corresponding substrings is also lower than the cutoff value; hence, the costly process of aligning the strings can be avoided.

In this section, we first describe the transformation from string space to integer space. Second, we explain the construction of the F-index, a data structure which groups nearby points, thereby allowing for further efficiency improvement. Third, we present an algorithm for computing an upper bound for the alignment score in frequency space. Finally, we give an algorithm for fast comparison of a set of points (corresponding to the substrings of a string) to an F-index built on another string.

Frequency space and F-index

Let s be a string from an alphabet Σ, and let $\sigma = |\Sigma|$. The frequency vector, f_s, of s is defined as the σ-dimensional vector whose entries are the number of occurrences of the letters in Σ (Kahveci and Singh, 2001). For DNA strings, the alphabet is $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}, \mathrm{N}\}$ (the letter N stands for unknown). Therefore, the frequency space of a DNA string has five dimensions. For example, if CTACCNTTAG is a DNA string, then its frequency vector is [2, 3, 1, 3, 1] ([#As, #Cs, #Gs, #Ts, #Ns]).

Two strings that are equal clearly have the same frequency vector. It also seems intuitively true that two strings that are similar have similar frequency vectors, but the exact nature of this similarity requires some explanation. It is instructive to note how the frequency vector of a string changes in response to the three basic edit operations: inserting a letter increases one element of the vector, and deleting a letter decreases one element; finally, replacing one letter with another increases one element of the frequency vector and decreases another. Based on these observations, we define a distance measure which we call frequency distance. The frequency distance $\mathrm{FD}(u, v)$ between two frequency vectors u and v is the minimum number of basic edit operations required to turn a string the frequency vector of which is u into a string the frequency vector of which is v.

It is clear that the frequency distance between two strings is a lower bound of the edit distance between them. (If the frequency distance between two strings is d, at least d edit operations are necessary to change one string into the other.) This can be used to prune a search for similar strings based on their frequency vectors as follows: for each string, we compute its frequency vector. Then, instead of computing the edit distance between two strings, just compute their frequency distance, which requires less computation and main memory. If the frequency distance is above a certain threshold, we know that the edit distance must also be above the threshold; only

if the frequency distance is below the threshold do we compare the actual strings.

In order to find similar regions between two genomes, one can compute, for each genome, the frequency vectors for all substrings that are of a certain length w, the window size. Where frequency vectors from the two genomes are close together, it indicates that the corresponding regions of the string may align well. The choice of w affects the precision and speed of the technique: a larger w leads to faster computation at the risk of missing short alignments. Experiments show that a w around 4000 produces almost-perfect results quickly. The problem with this approach is that a large number of frequency vectors are generated. To address this problem, we aggregate vectors from one genome into an index structure which we call the F-index.

An F-index is a set of σ-dimensional boxes. For a number of consecutive frequency vectors from the same string, we compute a box—their minimum bounding rectangle—and add it to the index. The maximum number of points in each box is termed its box capacity. Various strategies for choosing the box capacity will be discussed later; for now, we use a fixed box capacity of c. Then, the F-index built on a string s using window size w consists of $\lceil (|s| - w + 1)/c \rceil$ boxes. For each box in the F-index, we store only its lowest and highest coordinates, starting location of the first substring in the box, and the box capacity. As the box capacity increases, the number of boxes in the F-index decreases. Thus, the memory usage of the F-index goes down. However, boxes containing more points are generally larger, leading to more false hits. We use a box capacity of 1000 in our experiments.

Computing scores in frequency space

Because of its popularity, we use BLAST-style alignment scores instead of edit distances. The score of an alignment of two strings depends on three parameters: S_{match}, S_{mismatch} and S_{N}. The first two are used when two letters have a match or mismatch, respectively. The third is used when a letter is aligned to an unknown letter (N). Whereas S_{match} is positive, the other two are negative, S_{mismatch} more so than S_{N}.[1]

The score of the best alignment between a string q and the substrings of a string s can be found by dynamic programming, but this takes $O(|q| \cdot |s|)$ time. Instead, we propose a new distance function called the frequency score, $\text{FS}_w(v, B)$, to find an upper bound to the score of the best alignment in $O(\sigma)$ time, where $w = |q|, v$ is the frequency vector of q and B is the box that covers the frequency vectors of the substrings of s of length w. If $\text{FS}_w(v, B)$ is less than a user-defined cutoff, then the actual alignment score is also less than that cutoff, thus the costly dynamic programming can be avoided. The algorithm that computes $\text{FS}_w(v, B)$ is given in Figure 1.

[1]Mismatch penalties are assessed instead of gap penalties. This guarantees that the computed score is an upper bound, even though the transformation to the frequency domain discards information about the order of the letters in each window.

/* v : σ-dimensional integer point
B : σ-dimensional integer box of lower and higher coordinates $B.L$ and $B.H$.
w : window size used to construct B */

Procedure $\text{FS}_w(v, B)$
1. *inc* := *dec* := *sum* := 0;
2. for i from 1 to $\sigma - 1$:
   if $v[i] < B.L[i]$ then
      inc += $B.L[i] - v[i]$;
      sum += $B.L[i]$;
   else if $B.H[i] < v[i]$ then
      dec += $v[i] - B.H[i]$;
      sum += $B.H[i]$;
   else
      sum += $v[i]$;
3. *ScoreInc* := $(\min\{sum, w\} - inc) \cdot S_{\text{match}} + inc \cdot S_{\text{mismatch}}$;
4. *ScoreDec* := $(\min\{sum, w\} - dec) \cdot S_{\text{match}} + dec \cdot S_{\text{mismatch}}$;
5. if $w <$ *sum* then
      ScoreInc += $S_{\text{N}} \cdot (sum - w)$;
   else if *sum* $< w$ then
      ScoreDec += $S_{\text{N}} \cdot (w - sum)$;
6. return $\min\{ScoreInc, ScoreDec\}$;

Fig. 1. Procedure $\text{FS}_w(v, B)$ for computing the best score of the alignment between a string x and a set of strings χ, where v is the frequency vector of x and B is the box that covers the frequency vectors of the strings in χ.

The algorithm starts by finding the number of mismatches (steps 1 and 2). Variables *inc* and *dec* count the total number of increments and decrements one needs to perform on the entries of v to shift it into B. Variable *sum* stores the sum of the entries of v when it is shifted into B. This is repeated for each of the first $\sigma - 1$ dimensions; the σth dimension represents Ns, which will be dealt with in step 5. Since each increment or decrement of v corresponds to a mismatch, the upper bound on the total number of matches is computed by subtracting the number of mismatches from w, the number of letters in the aligned string. The scores for matches and mismatches are computed using the upper bound on the number of matches and the number of mismatches (steps 3 and 4).

Notice that *sum* $= w$ if there are no Ns, i.e. if the fifth dimensions are zero for both the point and the box. If *sum* $\neq w$, their difference is a lower bound on the number of alignments with N, and we assess the appropriate penalties in step 5. Finally, step 6 returns the smallest of the two computed upper bounds.

In the worst case, the distance computation takes 18 integer additions, 5 integer multiplications and 10 integer comparisons. This is negligible compared to classic dynamic-programming algorithms with quadratic time complexity in terms of the alignment length.

Constructing the match table

In order to align two genomes, we first create an F-index on one of them and extract a number of points (frequency vectors) from the other. The point extraction procedure is similar to the one used when constructing the F-index, but instead of sliding the window one letter at a time, we shift it by w letters. A string q therefore yields $\lceil |q|/w \rceil$ points.

The next step is finding all box-point pairs that match, meaning that their FS_w scores are above the cutoff value. Computing the FS_w score for all boxes of the first string and all points of the second string would take too long. Notice, however, that a point and a box with a high FS_w score appear close together in frequency space. We exploit this to reduce the number of FS_w computations by partitioning the frequency space and comparing the points and boxes in each partition separately.

The space is first partitioned along the first dimension. The resulting partitions are then partitioned again along the second, third and fourth dimensions. We do not partition along the fifth dimension because the corresponding letter, N, is rare. We use an equi-depth histogram-based technique (Piatetsky-Shapiro and Connell, 1984) to ensure that each partition contains an equal number of points.

For inexact matches to be detected reliably, we grow each box in each direction by an amount equal to the cutoff value before deciding which partitions it intersects. If this was not done, it would lead to false negatives, as points and boxes that are very close but in two different partitions would not be reported.

A higher number of partitions leads to fewer score calculations (and, consequently, shorter running time) because it, for a given box, eliminates the need to compute the score with points in partitions that are not intersected by the box. Figure 2 illustrates this: without partitioning, 12 score calculations would be necessary for the 3 points and 4 boxes shown, whereas 4 calculations are sufficient when the space is partitioned as shown in the figure. It is, for instance, not necessary to compute box B_1's distance to point p_2 because B_1 does not intersect the partition that contains p_2. Of course, beyond a certain number of partitions, partitioning again does not reduce the number of distance calculations much, and the overhead of handling the large number of partitions causes the running time to go up. With our datasets, about five partitions in each dimension ($5^4 = 625$ in total) gave the best results.

If a point and a box match, the corresponding parts of the two genomes have to be aligned. We record this in the match table, a data structure which (1) provides a memory-efficient representation of which substrings must be aligned and (2) is suitable for deciding on an optimal I/O schedule, as described in the next subsection.

Conceptually, each genome is divided into pages, i.e. non-overlapping blocks of a fixed size. This is because a page is the minimum I/O unit, i.e. at least one page is read from the disk at a time. Fixed-size pages are used because they allow for efficient I/O scheduling, as explained in the next subsection. A page is not the same as the substring of a box: the box capacity can be different from the page size, and the substrings corresponding to two boxes may be overlapping. There is a many-to-many relationship between boxes and pages: the substring corresponding to a box may span more than one page, and (parts of) a page may occur in the substrings of more than one box.

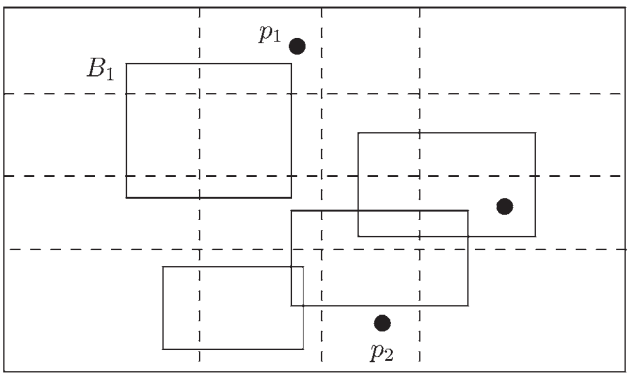

Fig. 2. Two-dimensional illustration of the four-dimensional frequency space containing boxes and points, partitioned by the dashed lines. Only three points and four boxes are shown.

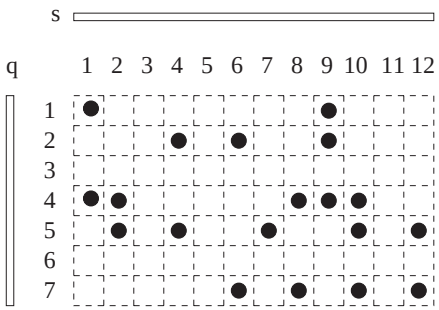

Fig. 3. Match table for two strings q and s. The black dots identify true entries.

The match table, illustrated in Figure 3, is a matrix with one column for each page of the first genome and one row for each page of the other. In this figure, q has 7 pages and s has 12 pages. Each entry of the match table contains either true or false, and uses one bit of storage. If a point and a box match, we mark as true all the entries in the rectangular section of the match table that corresponds to the substrings represented by the box and point.

Increasing the page size allows larger genome strings to be compared in the same amount of memory. For example, with a page size of 4 kb, it takes 1250 MB of main memory to compare two 400 Mb genomes, whereas by doubling the page size to 8 kb, two 800 Mb genomes can be compared in the same amount of space. On the other hand, a smaller page size produces a more precise match table, since each

entry corresponds to a smaller region in each string. In our experiments, we set the page size equal to the window size used in the construction of the F-index.

Our method essentially finds an upper bound to the alignment score of substrings of length w. We exploit this to find local alignment scores of longer strings by partitioning the strings into substrings. However, this strategy has two disadvantages. First, a similar query substring may be split into two partitions. Second, similarities between substrings of lengths less than the window size w may be missed. The former problem can be solved by extending the query substrings once the query is partitioned. The latter problem can be solved by using an appropriate window size.

THEORETICAL ANALYSIS

The F-index structure has a small memory footprint and provides fast distance computation in the frequency domain, but carries less information than the original string because the same frequency vector may correspond to multiple substrings. Therefore, searching in the frequency domain may result in false positives. In this section, we develop theoretical formulas to estimate the number of hits for the F-index.

We analyze the F-index structure in two steps. First, we estimate the box size for a given box capacity. Second, we predict the probability of intersection between two boxes, or between a box and a point.

Estimating box size

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$ be the alphabet of the dataset. Let $r = [r_1, \ldots, r_{|\Sigma|}]$ be the profile for the dataset, where r_i is the rate at which σ_i occurs. For example, the profile for a DNA dataset in which the letters are uniformly distributed is $r = [0.25, 0.25, 0.25, 0.25]$.

We will start by considering the dimension for a single letter in the alphabet, then extend the analysis to all the letters. Consider the frequency vector v corresponding to the initial positioning of the window on the dataset. Let $v_i = t$ be the i-th value of this vector (i.e. the number of σ_is in the first window). When the window is slid by one, there are three possibilities for v_i: it can decrease by one, stay the same or increase by one. This is shown using a state transition diagram in Figure 4. Here, each node corresponds to a state. The labels on the nodes represent the value of v_i. The labels on the edges are the probabilities of moving from one state to the other by sliding the window one position. As the window is slid, v_i traverses a set of consecutive nodes by moving right and left one node at a time.

We define $P_i(n, k, j)$, for $n \geq k \geq j \geq 1$, as the probability that sliding the window n times causes exactly k states to be traversed in the state transition diagram defined by the rate of letter σ_i, and that the final state is the j-th state relative to the leftmost of the k visited states.

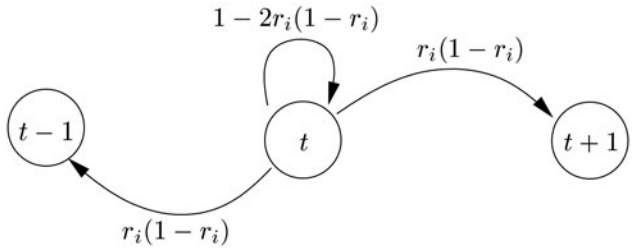

Fig. 4. The state transition diagram for the i-th entry of a frequency vector.

We define a recurrence function for $P_i(n, k, j)$ to be 1 if $n = 1$ and

$$
\begin{aligned}
&[1 - 2r_i(1 - r_i)]P_i(n - 1, k, j) && \{\text{Stay}\}\\
&+ r_i(1 - r_i)P_i(n - 1, k, j - 1) && \{\text{Move right}\}\\
&+ r_i(1 - r_i)P_i(n - 1, k, j + 1) && \{\text{Move left}\}\\
&+ r_i(1 - r_i)P_i(n - 1, k - 1, 1) && \{\text{Extend left}\}\\
&+ r_i(1 - r_i)P_i(n - 1, k - 1, k - 1) && \{\text{Extend right}\}
\end{aligned}
$$

if $n > 1$.

For a given n, the value of $P_i(n, k, j)$ can be easily calculated using dynamic programming for all $n \geq k \geq j \geq 1$. The number of states visited after n slides of the window, $L_i(n)$, corresponds to the width of the box along the i-th dimension. The distribution of this random variable can be computed as

$$P[L_i(n) = k] = \sum_{j=1}^{k} P_i(n, k, j).$$

Estimating hit ratio

We will estimate the probability that a given frequency vector of a query string overlaps with a given box in the F-index. Let w and c be the window size and box capacity used to create the F-index, and let v be the frequency vector of the query string for a given window position. Notice that $0 \leq v_i \leq w$ for all $1 \leq i \leq |\Sigma|$. Let r_i be the rate at which the letter σ_i occurs. Then, the probability $P(v_i = t)$ that the letter σ_i occurs t times in a random positioning of the window is

$$P(v_i = t) = \binom{w}{t} r_i^t (1 - r_i)^{w-t}.$$

$P(v_i = t)$ is a binomial distribution with rate r_i.

Let L_i be the size of a box and u_i be its mid-point along the i-th dimension. The probability that a box intersects a random frequency vector, v, along the i-th dimension is equal to $P(|u_i - v_t| \leq L_i/2)$. This probability is given by

$$P\left(|u_i - u_i| \leq \frac{L_i}{2}\right) = \sum_{t=0}^{w} \sum_{s=\max\{0,t-L_i/2\}}^{\min\{w,t+L_i/2\}} P(v_i = t)P(u_i = s).$$

This probability can be calculated quickly by approximating the binomial distribution $P(v_i)$ using the normal distribution with mean wr_i and variance $wr_i(1-r_i)$. This approximation is very accurate for large values of w, i.e. for $w \geq 10$ (Ewens and Grant, 2001).

Using this formula, the probability of intersection between a box and a point can be calculated as

$$\sum_{\substack{L_i \leq \min\{w,c\} \\ \Sigma_j L_j \leq 2c+\sigma}} \left[\prod_{i=1}^{|\Sigma|} P\left(|u_i - v_i| \leq \frac{L_i}{2} \right) \right] \cdot P(L_i).$$

Computing an estimation for the hit ratio according to this formula is very costly because the formula is over all possible box sizes. Therefore, we approximate it by computing the expected size of a box along each dimension in advance. After this approximation, the probability of intersection between a box and a point reduces to

$$\prod_{i=1}^{|\Sigma|} P\left(|u_i - v_i| \leq \frac{\mathrm{E}(L_i)}{2} \right).$$

Experimental validation of analysis

The theoretical formulas assume that the distribution of letters remains the same as the window is slid over the string. However, real data may be skewed so that different letters occur at various frequencies in different regions of a string.

We tested the accuracy of our formulas by running experiments on real DNA strings. We used the genetic code of *Escherichia coli* (K-12 MG1655, acc. U00096) and the entire human genome as the real datasets. *E.coli* and the human genome consist of 4.6 million and 2.7 billion bp, respectively.

Figure 5 plots the probability of intersection between a box and a point generated from the same file for different box capacities and window sizes. The probability of intersection drops as the window size increases and the box capacity decreases. This experiment shows that the skewness of the real data reduces the probability of intersection. Thus, the actual pruning obtained for real data is even better than estimated by our formula using uniform distributions. It is possible to use our formulas for other distributions, but that is unlikely to yield better predictions because the distribution function for the letters varies between different regions of real data.

The window size, w, is a parameter, and should be set to the length of the shortest alignment that is of interest, as alignments shorter than w are not detected reliably. For whole genome alignment we suggest using $w = 2$ or 4k.

SCHEDULING DISK I/O

Because the match table identifies all substring pairs that might be similar, only the substring pairs corresponding to entries that are marked as true need to be searched. For example, $M_{1,1}$ is marked (in Fig. 3), so the first page of the first string must be compared with the first page of the second string. Current string alignment tools start by finding seeds within one of the strings (for instance BLAST finds exactly matching substrings of length 11).

One could simply construct a hash table on one of the strings and sequentially scan the parts of the other string that correspond to marked rows or columns. This is an improvement over other search tools since the search is restricted to the marked entries. However, if both the strings are very large, the hash table may not fit into memory, resulting in excessive random disk I/O. In order to prevent this, we, iteratively, cut slices from the match table by splitting it either vertically or horizontally. As the match table is split along a direction, the corresponding string is also partitioned into shorter strings. Then, we construct a hash table on the marked pages of the unsplit string, and sequentially scan the other.

The decision on split direction is made as follows. Let r and c be the number of marked rows and columns of the match table. If $r < c$, then the match table is split vertically. Otherwise, the match table is split horizontally. The optimality of our dynamic splitting algorithm can be proved by considering the expected number of marked entries in each slice.

Figure 6 illustrates the slices obtained from a sample match table. In this example, we assume that the available memory can hold three pages. There are nine marked columns and five marked rows. Since the number of marked columns is larger than the number of marked rows, MAP chooses a vertical slice in the first iteration (shown with a rectangle in Fig. 6a). The slice has two properties: (1) it has at most three rows, and (2) it is as wide as possible. The first restriction ensures that the hash table built on the rows fits into memory. The second restriction forces MAP to process as many entries as possible at each iteration. The first slice has rows r_1, r_4 and r_5, and columns c_1 and c_2. BLAST constructs a hash table on the substrings of r_1, r_4 and r_5. Then it sequentially scans c_1 and c_2, and searches their contents within r_1, r_4 and r_5 using the hash table. When the search is complete, MAP removes the slice from the match table, and iterates on the rest of it. In the second iteration (Fig. 6b), the match table contains seven marked columns and five marked rows. Therefore, MAP splits the table vertically again. Note that MAP does not need to consider columns c_3 and c_5, as they do not contain any marked entries. The match table has four marked columns and five marked rows in the third iteration (Fig. 6c), so a horizontal split is chosen, and the hash table is constructed on the columns.

Once a slice is cut from the match table, we iterate through the split string on a slice-by-slice basis. A page from the split string is read only if its corresponding row or column contains at least one true entry in that slice. The pages are read using an optimal disk read schedule (Seeger, 1996). The optimal disk read schedule guarantees that the total disk I/O cost of reading candidate strings is never more than that of a sequential scan. For example, for the match table in Figure 3, after we cut a slice horizontally at the third row, we iteratively read q_1

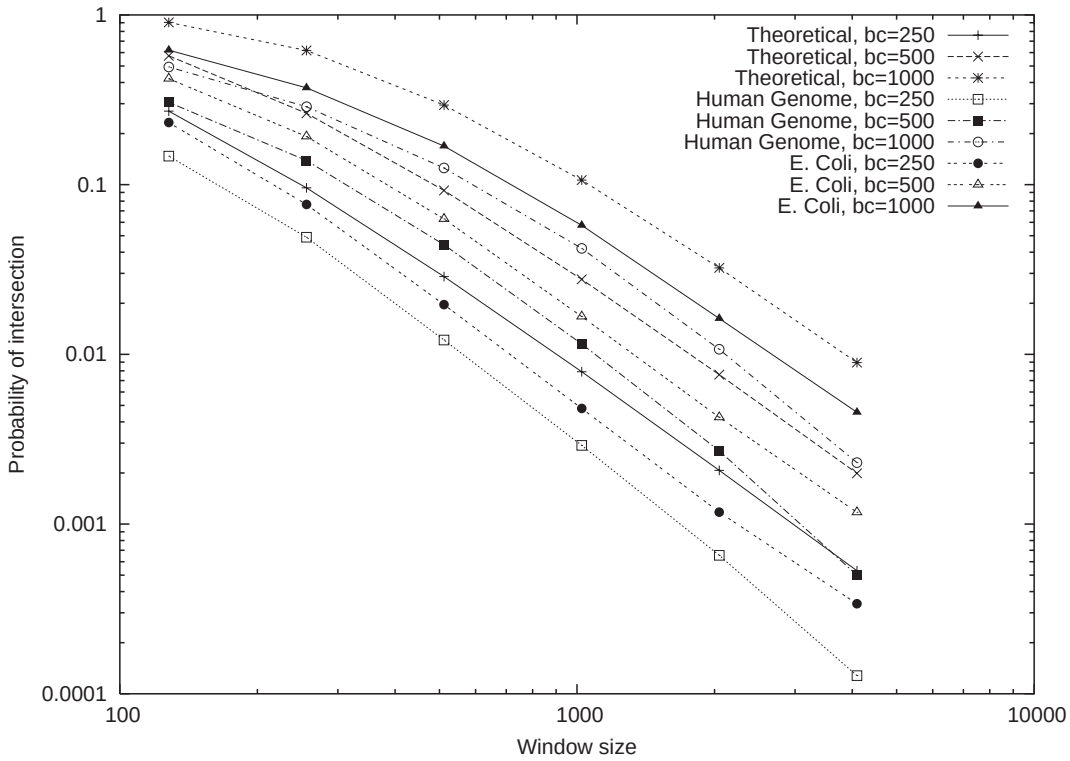

Fig. 5. The theoretical and experimental values for the probability of intersection of a box and a point. A box tightly encloses the frequency vectors of bc consecutive windows of length w. The experiments are performed on the genetic code of the *E.coli* bacteria and the entire human genome separately.

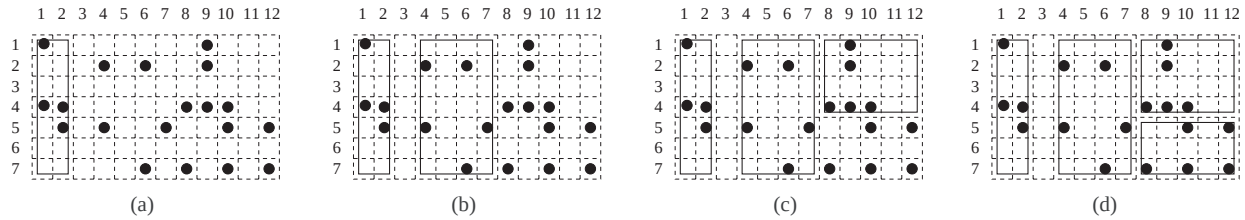

Fig. 6. The slices determined by the MAP algorithm. We assume that the available memory can store the hash table for at most 3 pages. (**a**) and (**b**) show vertical partitions whereas (**c**) and (**d**) show horizontal partitions.

and q_3, and search them using the hash table constructed on s_1 and s_4.

Figure 7 presents the pseudocode for the MAP algorithm. The inputs to the program are the match table, the match table boundaries and the amount of available memory. The algorithm starts by checking the boundaries. If the match table is all consumed, it quits (step 1). If there are still unprocessed regions, the number of marked rows and columns are computed first (steps 2 and 3). If the number of marked rows is less than the number of marked columns, the algorithm goes into vertical split mode (step 4); otherwise it switches to horizontal split mode (step 5). In vertical split mode, the splitting point is iteratively advanced column-wise unless the hash table for the slice fits into available memory (step 4a). The lower boundary for the columns of the match table is updated (step 4b). A hash table is then constructed on the marked rows of the slice (step 4c), and a search is performed by reading the marked columns of the slice using optimal disk scheduling (step 4d). Horizontal partitioning is performed in a similar fashion. The MAP search algorithm is then called recursively on the remaining match table (step 6).

EXPERIMENTAL EVALUATION

Our experiments, which assess the performance of MAP, were performed on computers with two AMD Athlon MP 1800+

```
/*  M                          : match table
    rowStart, rowStop          : row boundaries
    colStart, colStop          : column boundaries
    B                          : memory size          */
MAP(M, rowStart, rowStop, colStart, colStop, B)
  1. if ((rowStart > rowStop) or
         (colStart > colStop)), then
           Return; /* Match table is consumed. */
  2. numMarkedRow := number of marked rows in the range;
  3. numMarkedCol := number of marked columns in the
     range;
  4. if (numMarkedRow < numMarkedCol)
     /* Vertical partition */
       (a) For i := colStart to colStop
             - M' := M[rowStart : rowStop][colStart : i];
             - If B < hashTableSize(M', rows) then
                 If i > colStart then i--;
                 Break;
       (b) colStart := i + 1;
       (c) Construct hash table on marked rows of M';
       (d) Search marked columns of M' on the hash table;
  5. else /* Horizontal partition */
       (a) For i := rowStart to rowStop
             - M' := M[rowStart : i][colStart : colStop];
             - If B < hashTableSize(M', columns)then
                 If i > rowStart then i--;
                 Break;
       (b) rowStart := i + 1;
       (c) Construct hash table on marked columns of M';
       (d) Search marked rows of M' on the hash table;
  6. MAP(M, rowStart, rowStop, colStart, colStop, B);
```

Fig. 7. The MAP search algorithm. The algorithm takes a match table, the boundaries of the search region, and the available buffer size as input. Later the search region is partitioned and aligned based on these parameters.

processors and 1- or 2-GB main memory, running Linux 2.4.19. In all experiments, unless otherwise noted, the page size and window size were set to 4 kb, and the error rate to 1%.

Quality comparison

Figure 8(a) shows the match table constructed for aligning the genomes of two strains of *E.coli* (K-12 MG1655, acc. `U00096`; O157:H7, acc. `BA000007`). The diagonal run of marked entries depicts the similarity patterns. The genomes are about 5 Mb, but MAP created the match table in less than a second. Figure 8b–d show the match table for the same strings when one of the strings is altered by translocation of two substrings, inversion of one substring, and duplication of one substring, respectively. These figures show that our match table can locate translocations, inversions and duplications quickly without aligning the strings. Figures like these provide scientists with a quick visualization of the similarity patterns between two strings without actually aligning them.

Figure 9 shows a similar picture generated from the actual alignments of the two strains of *E.coli*, as computed by BLAST. Each alignment is represented by a line between the points corresponding to the alignment's start and end positions. The similarity to Figure 8(a) shows that the match table provides an accurate view of the similarities between the strings.

We also compared the complete genomes of two strains of *Streptococcus pneumoniae* (R6, `NC_003098`; TIGR4, `NC_003028`) with BLAST and MAP. The window size for MAP was set to 4 kB. In this experiment we observed that MAP and BLAST give results of similar quality, but that MAP misses parts of a few of the long alignments.

These are alignments that are cut in two by the slices of the match table. It is possible to stitch together the fragments in a postprocessing step.

Our final quality experiment quantifies the alignments found by BLAST but missed by MAP. We ran BLAST and MAP on two strains of *E.coli* (K-12 MG1655, acc. `U00096`; O157:H7, acc. `BA000007`). BLAST was run with default parameters, whereas MAP was run with $w = 4096$ and $\epsilon = 1$ and 2%. For each BLAST alignment, we computed the percentage of the alignment that was also found by MAP. For each score S in the BLAST result set, we computed the average of such percentages over all BLAST alignments scoring at least S. The results, plotted in Figure 10, show that MAP has a 100% recall for high-scoring alignments. As the score drops, so does MAP's recall. This is because lower-scoring alignments usually contain more mismatches and indels. Lower-scoring alignments can be detected reliably by running MAP with a higher ϵ-value, but this has the side effect of increasing the density of the match table, and thus MAP's running time. (In the worst case, i.e. when $\epsilon = 1.0$, all the entries of the match table get marked. In this case, MAP's speed becomes the same as that of BLAST.) Even for alignments as short as 4096, MAP's recall is more than 95 and 97% (for $\epsilon = 1$ and 2%, respectively). This experiment shows that MAP misses only a small percentage of BLAST's local alignments for similar strings. We achieved similar results in the comparison of two strains of *S.pneumoniae* (R6, `NC_003098`; TIGR4, `NC_003028`). However, MAP's recall varied between 75 and 100% for alignment of distant strings such as *Homo sapiens* 12p13 (`U47924`) and *Mus musculus* chromosome 6 (`AC0002397`).

Performance comparison

To test the performance of our MAP program, we measured the time to compute the match table for strings of length from 50 to 450 Mb. The strings were prefixes of the *M.musculus* genome (the chromosomes from NCBIs build 30 catenated). We used a static box capacity of 1000 points and a window size of $w = 4$k. Figure 11 shows that the time to compute the match table is less than quadratic. The match table for two 450 Mb strings is computed in only 270 s.

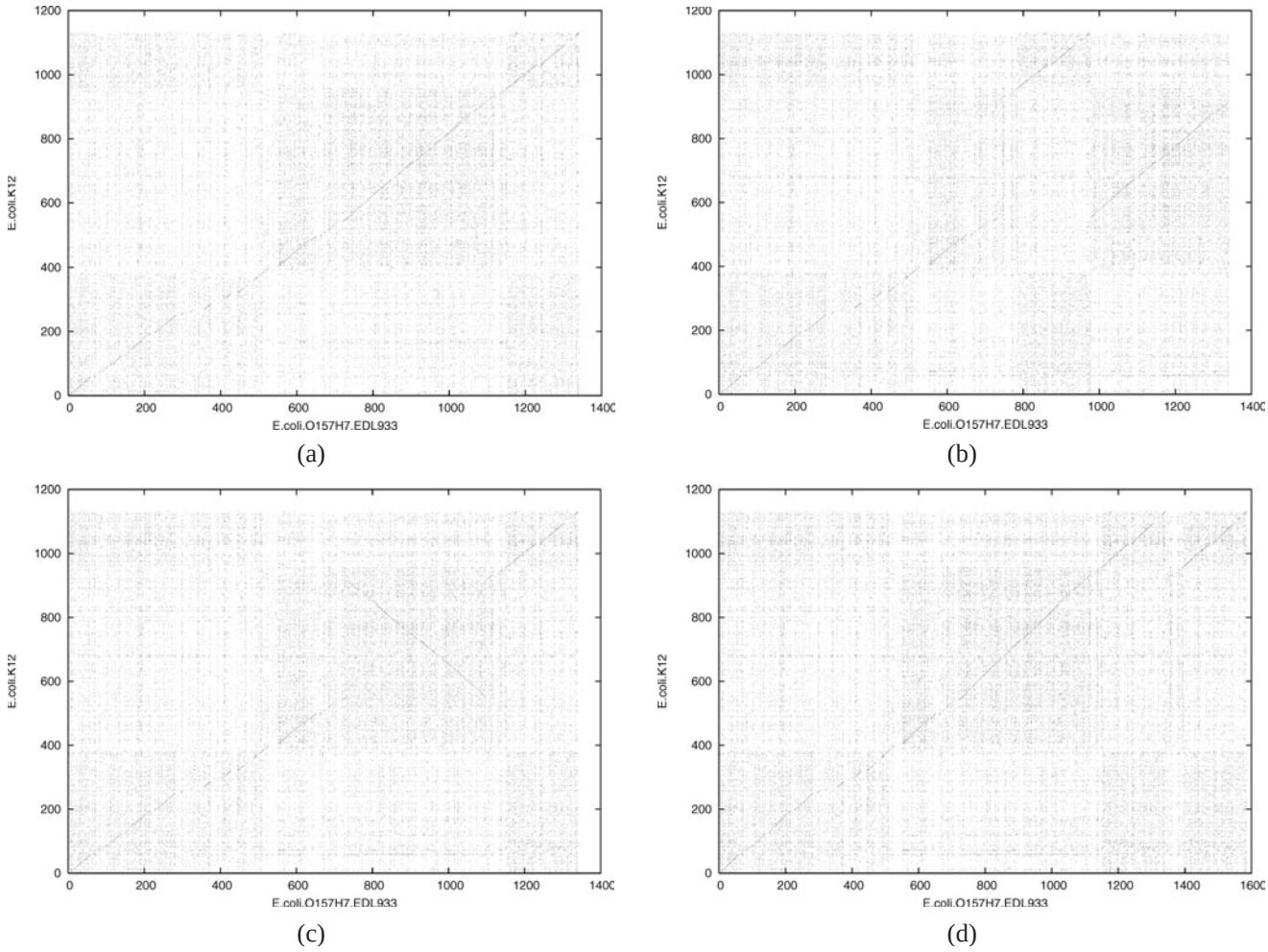

Fig. 8. (**a**) Match table created by MAP for aligning the genomes of two strains of *E.coli*. The points correspond to marked entries of the match table. Match table for the same strings when one of the strings is altered by (**b**) translocation of two substrings, (**c**) inversion of one substring, and (**d**) duplication of one substring.

The amount of memory required to compute a match table increases quadratically, from 20 MB for two 50 Mb strings to 1.5 GB for two 450 Mb strings. This is negligible compared to the size of the hash table a BLAST-like technique would construct: BLAST could not even align two strings of $<$20 Mb each on our 1-GB machine (Table 1).

Our match table achieved a high amount of pruning. On average, only 3.7% of the match table entries were marked in this experiment. We obtained slightly different pruning rates for two strains of *E.coli* (K-12 MG1655, acc. U00096; O157:H7, acc. BA000007). In this case, 7.5% of the entries were marked.

Constructing the match table is only one phase of MAP: the match table will also need to be sliced, and BLAST run on each slice. To compare BLAST and MAP, we aligned a number of strings, from 200 kb to 20 Mb, with both, and measured the memory consumption and running times. The results are shown in Table 1. For MAP, we used a static box capacity of 1000. BLAST did not complete the alignment of *Arabidopsis thaliana* chromosomes 2 and 4 in 10 days; nor could it align the much smaller *H.sapiens* chromosome 18 to itself in the same amount of time. The memory usage of BLAST for the last two lines of this table was $>$1 GB, the available memory on our computer. For short strings, the memory usage of BLAST is better than BLASTZ. However, as strings get longer, BLASTZ uses less memory than BLAST. The running time of BLASTZ is more than BLAST for the first six experiments. However, BLASTZ outperforms BLAST for the self-alignment of *H.sapiens* chromosome 18 and the alignment of chromosomes 2 and 4 of *A.thaliana*. This is because, unlike BLASTZ, the memory consumption of BLAST exceeds the amount of available memory.

The times for MAP in Table 1 are higher than necessary because the BLAST program being run on the partitions

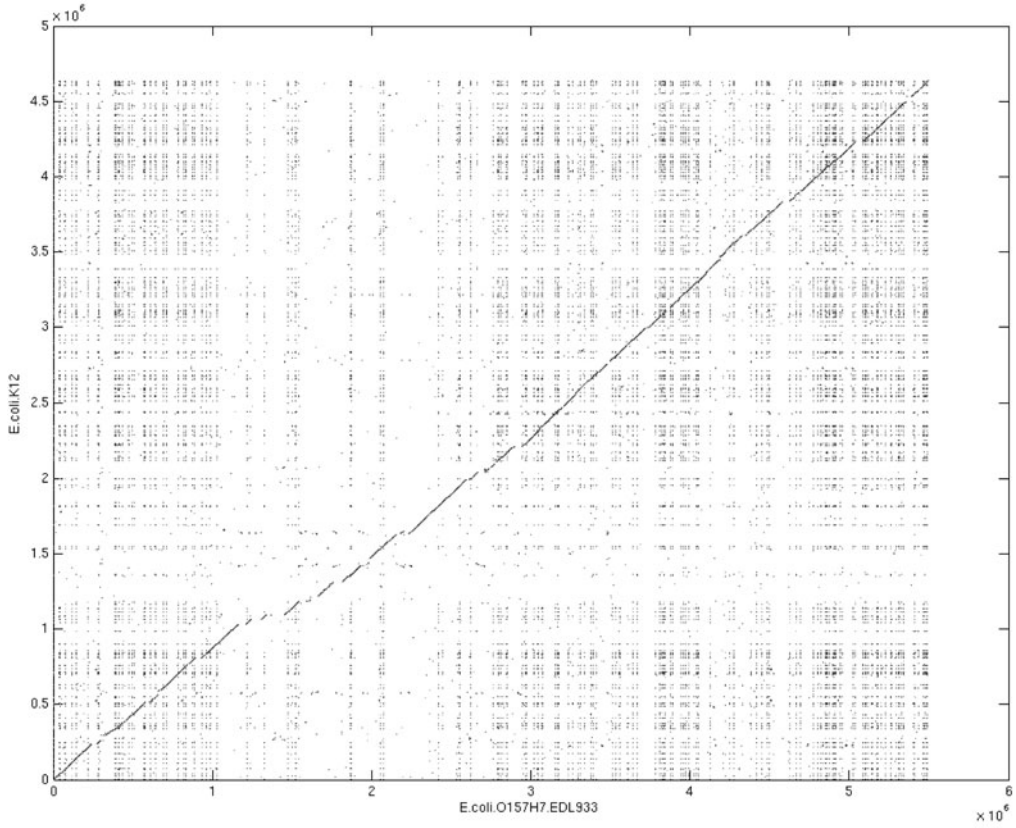

Fig. 9. The alignment of two strains of *E.coli* by BLAST.

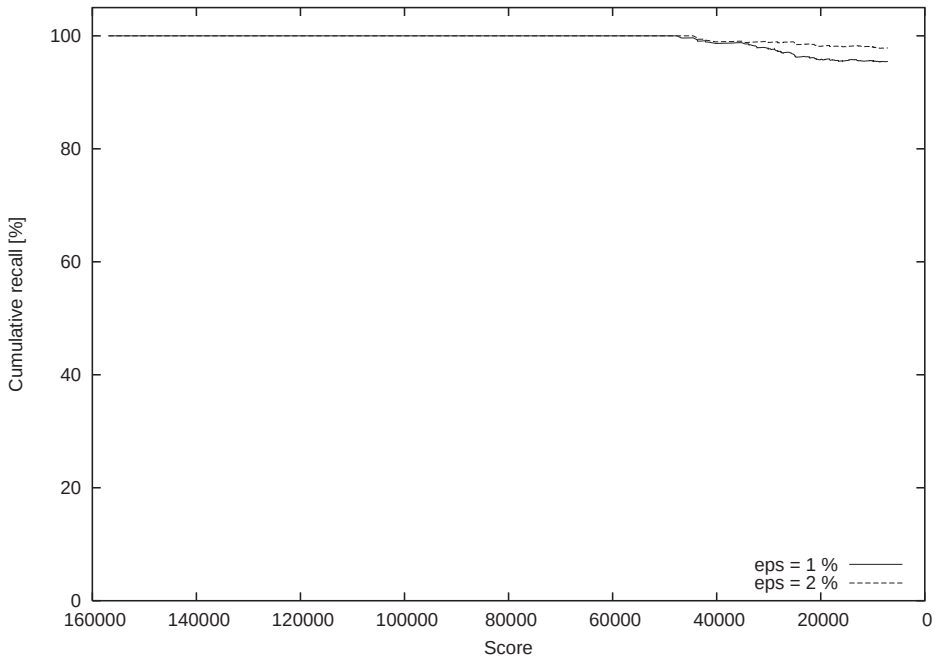

Fig. 10. The cumulative recall of MAP over BLAST's local alignments of length at least $w = 4096$ for $\epsilon = 1$ and 2% for the local alignment of two strains of *E.coli* (K-12 MG1655, acc. U00096; O157:H7, acc. BA000007).

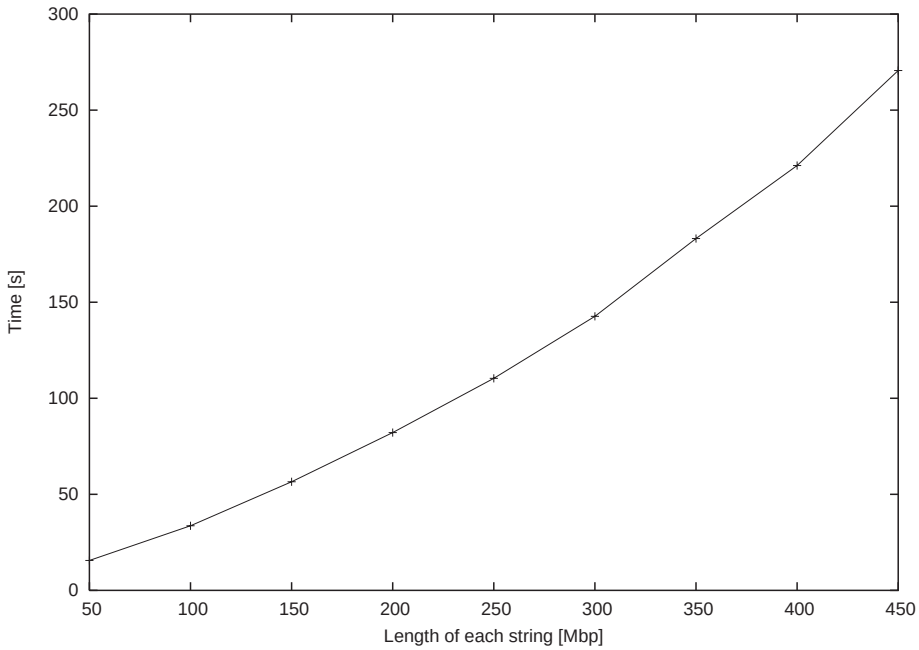

Fig. 11. Time to compute the match table for strings of different length.

Table 1. Running time and memory usage for BLAST and MAP for a number of smaller datasets

String 1		String 2		BLAST		BLASTZ		MAP		Match table	
Description	Len.	Description	Len.	Mem.	Time	Mem.	Time	Mem.	Time	Mem.	Time
H.sapiens 12p13[a]	0.22	*M.musculus* chr. 6[b]	0.22	16	3	74	36	6	2	0.06	0
M.genitalium[c]	0.57	*M.pneumoniae*[d]	0.8	46	6	108	27	29	5	0.1	0
S.pneumoniae R6[e]	2.0	*S.pneumoniae* TIGR4[f]	2.1	175	478	150	717	17	97	0.2	0
H.influenzae Rd[g]	1.8	*E.coli*[h]	4.6	290	45	140	105	89	31	0.38	0
M.tuberculosis CDC1551[i]	4.4	*M.tuberculosis* H37Rv[j]	4.4	320	456	1230	106,687	13	338	0.45	1
E.coli K-12 MG1655[h]	4.6	*E.coli* 157:H7[k]	5.5	380	614	263	827	17	364	0.53	1
H.sapiens chr. 18[l]	4.2	*H.sapiens* chr. 18[l]	4.2	920	∞	850	615,677	25	540	0.44	1
A.thaliana chr. 2[m]	19.9	*A.thaliana* chr. 4[n]	17.8	∞	∞	704	15,678	56	9870	3.6	5

The columns for MAP include running BLAST on the partitions. The match table columns include creating the match table. The units for string lengths, memory consumption, and time are Mb, MB and seconds respectively.
[a]U47924, [b]AC0002397, [c]L43967, [d]NC_000912, [e]NC_003098, [f]NC_003028, [g]L42023, [h]U00096, [i]AE000516, [j]AL123456, [k]BA000007, [l]NT_000864, [m]AC_006837, [n]AL_161471.

performs much more work than is needed. This is because, although BLAST can be instructed to limit itself to certain sections of the input, there is no way to specify that only the substring pairs corresponding to the marked entries in the match table should be searched. Thus, BLAST proceeds as if every entry in the partition were marked in our MAP implementation.

One way to reduce the memory consumption of BLAST is to partition one or both of the strings into substrings without any prior information about the strings and run a BLAST on each of these substrings. Our simple 'black box' implementation of MAP is still superior to this partitioning in a number of ways. First, partitioning the data based on the match table prunes large unmarked regions, thus it avoids searching the entire string. Second, the match table depicts a coarse-grained visualization of the actual alignment. Based on the diagonal runs and dense regions, a user may restrict the alignment to a smaller substring. This is important, especially for very long strings. For example, MAP computes the match table for two 450 Mb strings in only 270 s. If only a small percentage of bases are correlated in these strings, then the user can see this by inspecting the match table and align only those substrings.

To account for the time difference of MAP in Table 1, we performed an experiment where MAP, instead of handing the partitions to BLAST, handed them to a simulator. Based on the amount of memory available—a parameter of

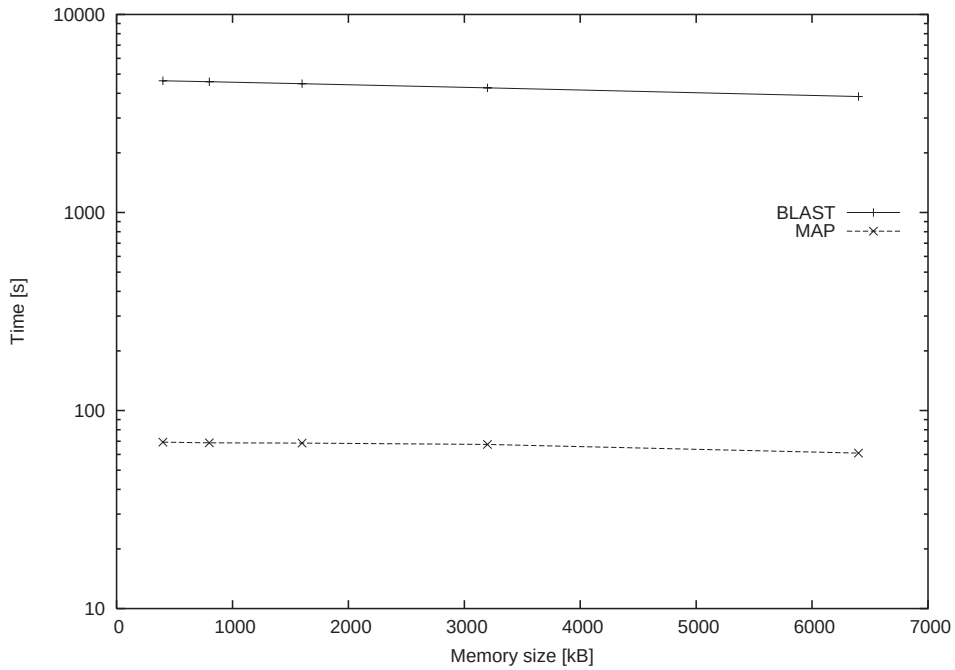

Fig. 12. The total time for BLAST and MAP to align two strains of *E.coli* (K-12 MG1655, acc. `U00096`; O157:H7, acc. `BA000007`). for different memory sizes.

the experiment—the simulator counted the number of disk seeks and reads necessary to align the marked entries in the partition. From these numbers and measurements of average seek times and read rates for our machine, we calculated the total cost. Figure 12 shows the total costs, according to this model, for aligning two strains of *E.coli* (K-12 MG1655, acc. `U00096`; O157:H7, acc. `BA000007`) with MAP and with BLAST. The total cost of MAP in this figure includes the cost of creating and partitioning the match table, in addition to aligning the strings. In this experiment, to BLAST's advantage, we set BLAST to construct the hash table on the shorter string (i.e. mouse chromosome 18). For the memory sizes considered, MAP performed up to two orders of magnitude better than BLAST.

Dynamic indexing strategies

So far, we have used a static box capacity in the construction of the F-index. One way to improve the F-index is to use a dynamic strategy based on the distribution of the frequency vectors. We implemented the following dynamic strategies:

- Fixed volume. Keep adding points to a box until its volume exceeds a certain threshold.
- Fixed density. Keep adding points until n/V, the number of points divided by the volume of the box, falls below a certain threshold.
- MHIST-Volume. Start with one big box containing all the points. Find the position to split the box such that the sum of the volumes of two new boxes is minimized, but the points in each box still correspond to consecutive positions of the sliding window on the string. Keep splitting the box with the largest volume to get the desired number of boxes.
- MHIST-Density. Like MHIST-Volume, but split a box in to two new boxes i and j such that the total density, $n_i/V_i + n_i/V_j$, is maximized. Keep splitting the box with lowest density.

The match table construction time for Volume and MHIST-Volume are 10–18% less than the time for the Static strategy. In terms of pruning rate, the Static, Volume and MHIST-Volume strategies were almost identical, and they gave the best results. On the other hand, the dynamic strategies require more time and space to construct the F-index. We recommend the use of the static strategy since the size of the index structure is smaller and its performance is very close to the best dynamic strategy.

We can summarize the experimental results as follows: (1) the output quality of MAP is very close to that of BLAST, (2) MAP is up to two orders of magnitude faster than BLAST and (3) MAP works well even for small memory sizes.

DISCUSSION

In this paper, we have considered the problem of finding local alignments for huge genome strings. We have presented an algorithm that computes scores in the frequency domain and an algorithm that finds local alignments between two strings.

The algorithm constructs a match table, a boolean matrix in which an entry is marked as true if the corresponding substrings may be similar, and false otherwise. The match table is the basis for pruning and for partitioning the search such that each partition can be searched in memory by an existing tool such as BLAST. The match table also serves as a visualization of the similarity between the strings prior to actual alignment.

In our experiments, we used BLAST (Altschul *et al*., 1990) for comparison. According to our experimental results, MAP runs up to two orders of magnitude faster than BLAST. Furthermore, MAP can work well even with long strings and little memory. MAP achieves these performance improvements while keeping the quality of the resulting answer set very close to that of BLAST.

MAP is a very general technique, in the sense that its match-table-based pruning and dynamic splitting scheme can be used to improve any of the current string search tools. Hence, one can view MAP as a technique that improves the available techniques instead of as a competitor to these techniques. It is also trivial to extend our method to global alignments. We solved a similar problem in Kahveci and Singh (2001).

Aligning large genome strings is an important emerging application. The explosive growth of these datasets and the complexity of computing matches makes it imperative that faster disk-resident techniques be devised. The techniques presented in this paper are an important step in this regard, and should be widely applicable.

ACKNOWLEDGEMENTS

This work was supported in part by grants EIA-0080134 and DBI-0213903 from the National Science Foundation.

REFERENCES

Altschul,S.F., Gish,W., Miller,W., Meyers,E.W. and Lipman,D.J. (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

Batzoglou,S., Pachter,L., Mesirov,J.P., Berger,B. and Lander,E.S. (2000) Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome Res.*, **10**, 950–958.

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

Brudno,M., Do,C., Cooper,G., Kim,M., Davydov,E., Program,N.C.S., Green,E., Sidow,A. and Batzoglou,S. (2003) LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA. *Genome Res.*, **13**, 721–731.

Brudno,M., Malde,S., Poliakov,A., Do,C., Couronne,O., Dubchak,I. and Batzoglou,S. (2003) Glocal alignment: finding rearrangements during alignment. *Bioinformatics*, **19**, 54i–62i.

Brudno,M. and Morgenstern,B. (2002) Fast and sensitive alignment of large genomic sequences. In *CSB*. Stanford, CA, August 2002, p. 138.

Burkhardt,S., Crauser,A., Ferragina,P., Lenhof,H.-P., Rivals,E. and Vingron,M. (1999) Q-gram based database searching using a suffix array (QUASAR). In *RECOMB*. ACM Press, France, pp. 77–83.

Califano,A. and Rigoutsos,I. (1993) FLASH: A fast look-up algorithm for string homology. In *ISMB*, Bethesda, MD, pp.56–64.

Delcher,A.L., Kasif,S., Fleischmann,R.D., Peterson,J., White,O. and Salzberg,S.L. (1999) Alignment of whole genomes. *Nucleic Acids Res.*, **27**, 2369–2376.

Ewens,W.J. and Grant,G.R. (2001) *Statistical Methods in Bioinformatics: An Introduction*. Springer, New York.

Gish,W. (1995) WU-BLAST. http://blast.wustl.edu/.

Gusfield,D. (1997) *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.

Huang,X. and Miller,W. (1991) A time-efficient, linear-space local similarity algorithm. *Adv. Appl. Math.* **12**, 337–357.

Kahveci,T. and Singh,A.K. (2001) An efficient index structure for string databases. In *VLDB*. Morgan Kaufmann, Roma, Italy, September 2001, pp. 351–360.

Kent,W.J. (2002) BLAT—the BLAST-like alignment tool. *Genome Res.*, **12**, 656–664.

Kurtz,S. and Schleiermacher,C. (1999) REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, **15**, 426–427.

Ma,B., Tromp,J. and Li,M. (2002) PatternHunter: faster and more sensitive homology search. *Bioinformatics*, **18**, 440–445.

Morgenstern,B. (1999) DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, **15**, 211–218.

Morgenstern,B., Frech,K., Dress,A. and Werner,T. (1998) DIALIGN: finding local similarities by multiple sequence alignment. *Bioinformatics*, **14**, 290–294.

Myers,E.W. (1986) An $O(ND)$ difference algorithm and its variations. *Algorithmica*, **1**, 251–266.

Needleman,S. and Wunsch,C. (1970) A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Mol. Biol.*, **48**, 443–453.

Pearson,W. and Lipman,D. (1988) Improved tools for biological sequence comparison. *Proc. Natl Acad. Sci., USA*, **85**, 2444–2488.

Piatetsky-Shapiro,G. and Connell,C. (1984) Accurate estimation of the number of tuples satisfying a condition. In *SIGMOD*, ACM Press. Boston, MA, pp. 256–276.

Schwartz,S., Zhang,Z., Frazer,K.A., Smit,A., Riemer,C., Bouck,J., Gibbs,R., Hardison,R. and Miller,W. (2000) PipMaker—a web server for aligning two genomic DNA sequences. *Genome Res.*, **10**, 577–586.

Seeger,B. (1996) An analysis of schedules for performing multi-page requests. *Inform. Syst.*, **21**, 387–407.

Smith,T. and Waterman,M. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.

States,D.J. and Agarwal,P. (1996) Compact encoding strategies for DNA sequence similarity search. In *ISMB*. AAAI, St Louis, MO, June 1998, pp. 211–217.

Tatusova,T.A. and Madden,T.L. (1999) BLAST 2 SEQUENCES, a new tool for comparing protein and nucleotide sequences. *FEMS Microbiol. Lett.*, **174**, 247–250.

Zhang,Z., Schwartz,S., Wagner,L. and Miller,W. (2000) A greedy algorithm for aligning DNA sequences. *J. Comput. Biol.*, **7**, 203–214.