MAP: SEARCHING LARGE GENOME DATABASES

TAMER KAHVECI AMBUJ SINGH
Department of Computer Science
University of California
Santa Barbara, CA 93106
{tamer,ambuj}@cs.ucsb.edu

Abstract

A number of biological applications require comparison of large genome strings. Current techniques suffer from both disk I/O and computational cost because of extensive memory requirements and large candidate sets. We propose an efficient technique for alignment of large genome strings. Our technique precomputes the associations between the database strings and the query string. These associations are used to prune the database-query substring pairs that do not contain similar regions. We use a hash table to compare the unpruned regions of the query and database strings. The cost of the ensuing search is determined by how the hash table is constructed. We present a dynamic strategy that optimizes the random disk I/O needed for accessing the hash table. It also provides the user a coarse grain visualization of the similarity pattern quickly before the actual search. The experimental results show that our technique aligns genome strings up to 97 times faster than BLAST.

1 Introduction

The growth in the amount of genomic information has spurred increased interest in large scale comparison of genetic strings. Conditions such as skin and colon cancer can already be classified into much finer categories than before and soon the same approach may be possible for heart disease, schizophrenia and many other conditions. Using this kind of genetic information helps to target the treatments at the precise form of the illness that has been diagnosed, thus helping patients and doctors weigh the risk and benefits of different treatments.

One important emerging application, called *Comparative Genomics*, analyzes and compares the genetic material of different species. It is the most reliable way to identify genes and predict their functions. The functions of the higher level organisms, like humans, can be revealed by comparing to their counterparts in similar or lower level organisms. Such genome analysis involve comparison of huge strings, as large as the whole genome of a species.

Phylogenetics and evolutionary studies are other important applications that use complete genetic information of different species. Phylogenetics is used to infer the ancestral relationships among different species. This sort of relations can be captured by comparing the genetic code of the whole genomes.

The amount of biological data has been growing exponentially. This growth is on a collision course with current homology search and database query techniques and presents new challenges to biological database design. Queries (as mentioned above) will be large and complex, databases will be huge, some data will be on disk, and significant portions of datasets will be local because of networking bottleneck and proprietary data. Fast and sensitive homology search algorithms are needed to 1) answer large queries, 2) handle huge databases stored on disk, and 3) interact with multiple datasets seamlessly.

In this paper, we propose an efficient algorithm for alignment of large genome strings. Our algorithm constructs a boolean *Match Table* for a given query string and database string with the help of the MRS index structure [1] (an index structure that stores the summary information for strings). The size of the MRS index structure is approximately 1-2% of that of database. Each entry of the Match Table corresponds to a query/database substring pair. An entry in the Match Table is marked as *True* if the corresponding query substring and database substring potentially contain similar patterns. It is marked as *False* otherwise. The size of the Match Table is negligible compared to that of database (typically 0.1% of the database.).

Once the Match Table is computed, we build hash tables on these strings as follows. First, we find the number of marked rows and columns by projecting all the *True* entries of the Match Table to its rows and columns. If the number of marked columns is more than the number of marked rows, we choose a vertical slice of the Match Table, and construct the hash table on the string that corresponds to rows of the Match Table. Otherwise, we choose a horizontal slice, and construct the hash table on the string that corresponds to columns of the Match Table. The width of these slices is restricted by available memory: the size of the hash table for a slice is no more than the available memory.

Once the hash table of a string for a slice is constructed, the marked substrings of the other string are read sequentially and exactly matching substrings (i.e. seeds) of the prespecified size (i.e. 11) are found using this hash table. The seeds are then extended in both directions to find better matches. Later, the results are reported in a descending score order. We call this technique *MAP* (MAtch table based Pruning).

The experimental results show that, MAP runs up to 97 times faster than BLAST [2] without decreasing the output quality. Furthermore, MAP can work well even for small memory sizes. This drastic reduction in CPU and I/O cost has the potential of making homology searches viable on desktop PCs. The filtering and scheduling techniques of MAP can easily be used to speedup and reduce the memory requirements of any of the current genome alignment tools. MAP also provides the user a coarse grain visualization of the similarity pattern between the strings prior to actual search.

2 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 [3,4]. For large data and query strings, this technique becomes infeasible in terms of both time and space. Myers [5] improved the time and space complexity to $O(rn)$ by maintaining only the required part of the distance matrix, where r is the amount of allowed error. However, for large error rates r is $O(m)$, and hence the complexity is still $O(mn)$. SIM [6] uses dynamic programming technique to find all the alignments. However, it is extremely slow for large database/query strings. GLASS [7] accelerates dynamic programming solution by finding exactly matching long substrings first. However, the time and space complexity of GLASS is still high since it requires the extraction of k-mers.

Many heuristic based search tools are developed to perform string alignment faster. We consider these tools in two categories: 1) hash table based 2) suffix tree based.

Some of the important hash table based tools are FASTA [8], BLAST [2], MegaBLAST [9],

BL2SEQ[10], WU-BLAST[11], SENSEI[12], FLASH[13], PipMaker (BLASTZ)[14], and PatternHunter[15]. These techniques are similar in spirit: they construct a hash table on either the query string, or the database string (or both) for all possible substrings of a prespecified size (say l). The value of l varies for these search tools and for different applications (e.g. BLAST uses l = 11 for nucleotides and l = 3 for proteins.). They start by finding exactly matching substrings of length l using this hash table. These exactly matching substrings are also called *seeds*. In the second phase, these seeds are extended in both directions, and combined if possible, in order to find better alignments. The main difference between these search tools is that they use different seed lengths and different seed extensions strategies. FLASH and PatternHunter also differs from them by choosing non-contiguous seeds. Current hash table based search tools handle short queries well, but become very inefficient in handling long queries in terms of both time and space.

There are also a number of homology search tools based on suffix trees (see [16] for suffix tree algorithms). These include MUMmer[17], QUASAR[18], and REPuter[19]. QUASAR builds a suffix tree on one of the strings, and counts the number of exactly matching seeds using this suffix tree. If the number of seeds for a region is more than a prespecified threshold, this region is searched using BLAST. REPuter builds a suffix tree on a string to find repetitions on the fly. MUMmer builds the suffix tree on both of the strings to find maximal unique matches. These tools are suitable for highly similar sequences. The main problems with suffix tree approach are twofold: (a) Suffix trees are inefficient at managing mismatches. This approach is perfect for highly similar sequences but fails to recognize more distant homologies. MUMmer and QUASAR implement various ways of linking up neighboring precisely matched blocks while REPuter lets user define a constant edit distance (default value is 3 at REPuter web site), the computation time increases very fast as the edit distance allowed increases. (b) Suffix trees have large space overhead. For example, the suffix tree in MUMmer costs $37n$ bytes of memory, where n is the input length[17], although with careful implementation, this can be reduced to $8n$.

All of the above mentioned tools require the use data structures larger than the database. Some of them are even larger than 200 times the database size. Extensive memory requirements make these tools infeasible for large scale genome comparison. Unlike these tools, the size of the MRS index structure is less than 2% of the database size. We will employ the MRS index structure in our technique.

3 Our contribution

3.1 MRS index structure

Let s be a string from an alphabet Σ, where $|\Sigma| = \sigma$. 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. The frequency vector of a DNA string has 4 dimensions. For example, if s = CTACCCTTAG is a DNA string, then $f(s) = [2, 4, 1, 3]$. A single edit operation on a string has one of the following effects on the frequency vector of that string: 1) Increase the frequency of a character by one (insert operation). 2) Decrease the frequency of a character by one (delete operation). 3) Increase the frequency of a character by one and decrease another by one (modify operation). Based on this fact, the *frequency distance* ($FD(f(q), f(s))$) between the frequency vectors of a query string q and a database

string s is defined as the minimum number of increments and/or decrements in order to transform $f(q)$ to $f(s)$.

Each frequency vector defines an equivalence class of a set of strings whose frequency vectors are equal. The frequency distance between two frequency vectors is equal to the minimum possible edit distance between two strings in their equivalence classes. As a result, if $r < FD(f(q), f(s))$, then we can conclude that the edit distance between q and s is also greater than r. There are no false drops and indels (insertion/deletions) are also handled. The frequency distance for DNA strings can be computed in constant time using 8 integer additions and 4 integer comparisons (see [1]).

Let $S = \{s_1, s_2, ..., s_d\}$ be a database consisting of potentially long strings from alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$. Let $w_1 = 2^a$ be the length of the shortest possible query string. The MRS index structure stores a grid of trees $T_{i,j}$, where i ranges from a to $a + l - 1$, and j ranges from 1 to d. The parameter l represents the number of resolution levels available in the index structure. Tree $T_{i,j}$ is the index structure for the j^{th} string corresponding to window size 2^i.

Tree $T_{i,j}$ is obtained by sliding a window of length 2^i on string s_j starting from the leftmost point of s_j. For each possible placement of the window, the frequency vector of the corresponding substring of s_j is computed. Note that each substring corresponds to a point in the σ dimensional integer space. The frequency vectors of the c consecutive windows are covered using a *Minimum Bounding Rectangle* (MBR), where c is the *box capacity*. Typically, the box capacity ranges between 1000 and 10000. Note that, only the lower and the higher end points of the MBRs are stored along with the starting locations of the first substring contained in that MBR (i.e. the frequency vectors are not stored.). Since the index structure considers frequencies at different resolutions, this index structure is called as the *Multi Resolution String (MRS) Index Structure* [1]. The MRS index structure can be constructed using a single sequential scan on the database. Therefore, the time complexity for index construction is $O(|D|)$, where $|D|$ is the database size.

3.2 Computing scores in the affine gap model

The MRS index structure was originally used for string comparisons using edit distance [1]. In this paper, we extend it to work with scores and affine model for gaps. An upper bound to the score of the best alignment (in the affine gap model) of a query string q to the strings contained in an MBR, B, of the MRS index structure can be approximated efficiently. This algorithm is shown in Figure 1. The algorithm takes a frequency vector v and a box B as input. It starts by finding the number of mismatches (Steps 1 and 2). Later, the scores for matches and mismatches are computed (Steps 3 and 4), and the cost of insertions/deletions are added (Steps 5 and 6). Note that the algorithm assumes a single gap for all insertions and deletions in order to maximize the score. In the worst case, $3 \cdot \sigma + 7$ integer additions, 5 integer multiplications and $\sigma + 1$ integer comparisons are sufficient to compute $FS_w(v, B)$, where σ is the alphabet size. For DNA strings this number is 19 integer additions, 5 integer multiplications, and 5 integer comparisons regardless of the length of the strings. Hence the cost of computing FS_w is negligible compared to classic dynamic programming based algorithms with quadratic time complexity.

3.3 Match Table construction

The Match Table is the crucial element of the proposed search technique. It computes local similarities of a query string and a database string using the approximation function (FS_w) on the MRS index structure. Unlike the indexes of the genome search tools discussed above, the size of the MRS index structure is negligible compared to database size. Furthermore, index searching using our upper bounding function for scores in Figure 1 is extremely fast.

As shown in Figure 2, the Match Table M for a query string q and a database string s, is a boolean matrix. It is constructed by performing a search on the MRS index structure. Each column of the Match Table corresponds to the substrings contained in an MBR of the MRS index structure. For example, if the box capacity is 1000, then the first column corresponds to the first 1000 substrings of the database, the second column corresponds to the second 1000 substrings, and so on. For simplicity, we set the box capacity to the page size. A number of subqueries are constructed by sliding a window of length w on query q with a shift amount of Δ. Each row of M corresponds to a set of consecutive subqueries of total size equal to a page size. In the figure, seven subqueries q_1, q_2, $\cdots$, q_7, each having a size equal to a page size, have been shown. Entry $M_{i,j}$ is set to *True* if the MBR B_j (or equivalently s_j) potentially contains a similar substring to the query substring q_i. A row/column is called *marked* if at least one of the entries in that row/column is set to *True*. For example, first and second rows in this figure are marked, but third row is not marked.

Figure 3 presents the algorithm used to construct the Match Table. The algorithm takes a query string q, an error rate ϵ, and a shift amount Δ as input and constructs their match table M. The algorithm starts by initializing M to *False* (Step 1). Later, the largest resolution in the MRS index structure that is less than $|q|$ (Step 2) is determined. A *cutoff threshold*, τ is determined using this resolution and the error rate (Step 3). A window of the specified resolution, w, is then slid on q with a shift amount of Δ, and the match table is constructed by comparing the frequency distance between the query substring and the MBRs to the cutoff threshold (Step 4). The default value for ϵ is 0.1 and for Δ and w is 4K. The time complexity of the *FIND_MATCHES* algorithm is $O(\frac{|q|-w}{\Delta} \cdot \frac{|s|-w}{c})$, and the space complexity is $O(\frac{|q|-w}{P} \cdot \frac{|s|-w}{P})$, where c is the box capacity of MBRs, P is the page size, and $|s|$ is the total database size. This time complexity can be reduced to $O((\frac{|q|-w}{\Delta} + \frac{|s|-w}{c}) \cdot log(\frac{|q|-w}{\Delta} + \frac{|s|-w}{c}))$ using a plane-sweep algorithm. High values of c ($c = P$) make these negligible compared to other costs.

3.4 Scheduling disk I/O

Once the match table is constructed, only the query/database page pairs that are marked as *True* need to be searched. For example, since $M_{1,1}$ is marked (in Figure 2), the first page of the query string (i.e. q_1) must be compared to the fist page of the database string (i.e. s_1). In order to do this, we need to find the seeds (i.e. exactly matching substrings of length 11) within s_1 corresponding to q_1 or vice versa.

One can simply consider constructing hash table on one of the strings and sequentially scanning the other string on the marked rows or columns. This is an improvement

/* v is σ dimensional integer point. */
/* B is σ dimensional integer box of lower and higher coordinates $B.L$ and $B.H$. */
Procedure $FS_w(v, B)$

1. $inc := dec := sum := 0$;
2. for i:= 1 to σ
   - 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. $ScoreInc :=$
   $(min\{sum, w\} - inc) \cdot S_{match}$
   $+ inc \cdot S_{mismatch}$;
4. $ScoreDec :=$
   $(min\{sum, w\} - dec) \cdot S_{match}$
   $+ dec \cdot S_{mismatch}$;
5. if $w < sum$ then
   $ScoreInc$ +=
   $S_{gapopen} \cdot (sum - w - 1)$
   $+ S_{gap\_extend}$;
6. else if $sum < w$ then
   $ScoreDec$ +=
   $S_{gapopen} \cdot (w - sum - 1)$
   $+ S_{gap\_extend}$;
7. return $min\{ScoreInc, ScoreDec\}$;

Figure 1: Procedure $FS_w(v, B)$ for computing the best score of the alignment between a string x and a set of strings $\mathcal{X}$, where v is the frequency vector of s and B is the MBR that covers the frequency vectors of the strings in $\mathcal{X}$.

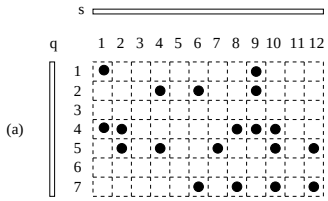


Figure 2: An illustration of Match Table on query q and database string s. The black dots correspond to *True* entries.

$/* INPUT$ q : query sequence
ϵ : error rate
Δ : shift amount
$OUTPUT$ M : match table $*/$

1. initialize M to *False*;
2. $w := 2^{r_{max}}$; /* the largest resolution in the index structure which is less than $|q|$. */
3. $\tau = w \cdot (1 - \epsilon)$; /* Compute cutoff threshold */
4. $start := 0$; $stop := |q| - w$;
5. While $start < stop$
   (a) $i = \lfloor (start + w)/pageSize \rfloor$;
   (b) $q' = q[start : start + w]$;
   (c) For each MBR B_j
   - If $FS_{r_{max}}(f(q'), B_j) \leq \tau$ then
     $M_{i,j}$:= True;

   (d) $start$ += Δ;

Figure 3: Construction of Match Table.

over currently available string search tools since the search is restricted to the marked entries. However, if both of the strings are very large, even the hash table of one of strings on marked rows/columns may not fit into the memory. This scheme may still cause excessive amount of random disk I/Os. In order to prevent this, we iteratively cut slices from the match table by splitting it either vertically or horizontally. Later, we construct a hash table on the marked pages of the unsplit string, and sequentially scan the other string.

Figure 4 illustrates the slices obtained from a sample Match Table by MAP. In this example, we assume that the available memory can store the hash table for 3 pages at most. The Match Table contains 9 marked columns and 5 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 (Figure 4(a)). This slice is shown using a rectangle. The slice has two properties: 1) It has at most 3 rows. 2) It is as wide as possible. The first restriction is imposed to ensure the hash table built on the rows fit into memory. 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. MAP constructs a hash table on the substrings in r_1,r_4, and r_5. Later, it sequentially scans c_1 and c_2, and searches their contents within r_1,r_4, and r_5 using the hash table. Once this search is completed, MAP removes this slice from the Match Table, and iterates on the rest of the Match Table. In the second iteration (Figure 4(b)), the match Table contains 7 marked columns and 5 marked rows. Therefore, MAP splits the table vertically. Note that, MAP does not need to consider columns c_3 and c_5 since they do not contain any marked entries. Match Table has 4 marked columns and 5 marked rows in the third iteration (Figure 4(c)). Therefore, it is a horizontal split in this step: hash table is constructed on the columns.

The decision for 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. For example, for the Match table in Figure 2, $r = 5$ and $c = 9$. Therefore, it is split vertically.

The size of a slice is determined based on memory size: The size of the hash table for the marked substrings of slice can not be more than the available memory. This can be determined by iteratively incrementing the split location. For example, consider the Match Table in Figure 2. Let memory size be 25 pages. Assume that the size of the hash table for a page is 8 times the size of a page (i.e. 2 words per letter). If the Match Table is split at the second column, then the slice contains three marked rows (r_1, r_4, and r_5). Hence, the memory requirement for this slice is $8 \cdot 3 + 1 = 25$ pages (here, the additional one page is reserved to store a page sequentially read from the other string). If we advance the cut to row r_4, the slice would contain four marked columns (r_1, r_2, r_4, and r_5), causing a memory requirement of $8 \cdot 4 + 1 = 33$ pages, which is more than available memory. Therefore, we decide to cut the Match Table at the second row.

Once a slice is cut from the Match Table, we iterate through the split string on an MBR by MBR basis. A page from the split string is read only if its corresponding row/column contains at least one *True* entry in that slice. The pages are read using an optimal disk read schedule [20]. The optimal disk read schedule guarantees that 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 2, once we cut a slice horizontally at third row,

we iteratively read q_1 and q_3, and search them using the hash table constructed on s_1 and s_4. The optimality of our dynamic splitting algorithm can be proved by considering the expected number of marked entries in each slice.

Figure 5 presents the pseudocode for the MAP algorithm. The inputs to the program are Match Table, Match Table boundaries, and the size of the 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 more (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 4.a). The lower boundary for the columns of the Match Table is updated (Step 4.b). A hash table is then constructed on the marked rows of the slice (Step 4.c), and a search is performed by reading the marked columns of the slice using optimal disk scheduling (Step 4.d). Horizontal partitioning is performed similar to vertical partitioning. The MAP search algorithm is then recursively called on the remaining Match Table (Step 6).

4 Experimental Evaluation

For our experimental evaluation, we used human chromosomes 18 and 21, mouse chromosome 18, E.coli.K12, and E.coli.O157H7.EDL933 taken from NCBI. These chromosomes are composed of the alphabet $\Sigma = \{A, C, G, T\}$. The human chromosome 18 contains 4M characters, human chromosome 21 contains 33M characters, mouse chromosome 18 contains 2.3M characters, E.coli.O157H7.EDL933 contains 5.5M characters, and E.coli.K12 chromosome 18 contains 4.6M characters. We downloaded the source code of the BLAST program, and implemented the MRS index structure. We ran our experiments on a 400 MHz Pentium II computer with 1 GB memory. Page size is set to 4K in all experiments.

4.1 Quality comparison

Our first experiment set compares the quality of outputs. For this experiment, we generated five query sets from human chr18 dataset for $|q| = \{500, 1000, 2000, 4000, 8000\}$. Each of these query sets contains 100 queries. Later, we modified these queries with 0.05 mutation probability using three edit operations (i.e. insert, delete, and modify). We performed queries using these five query sets on human chr18 dataset using BLAST and MAP with quality cutoff values of $\epsilon = \{0.1, 0.2, 0.25\}$.

Figure 6 plots the score of the first 1000 outputs of BLAST and MAP for $\epsilon = \{0.1, 0.25\}$ for $|q|$ = 4000, when w is chosen as the maximum possible resolution in the index structure and $\Delta = w$. The results for other query lengths were similar. The quality of MAP outputs are close to those of BLAST even when $\epsilon = 0.1$. The deviation between BLAST outputs and MAP outputs are about 5% for high scoring outputs, and 10-30$\%$ for low scoring outputs when $\epsilon = 0.1$. For $\epsilon = 0.25$, this deviation is almost zero for high scoring outputs and 5% for low scoring outputs.

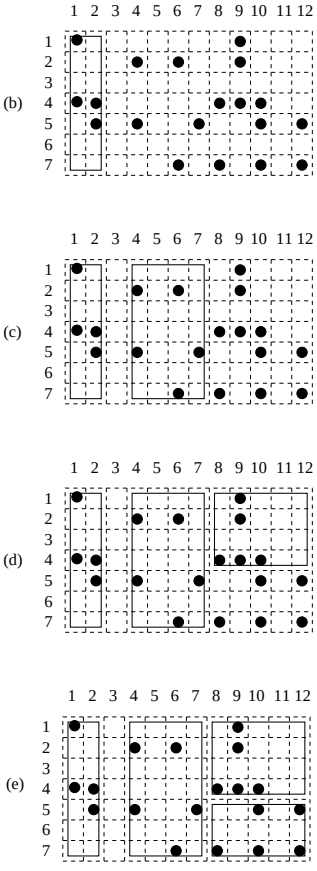


Figure 4: The slices determined by the MAP algorithm. We assume that the available memory can store the hash table for 3 pages at most.

$/* \; INPUT \quad M : match\ table$
$rowStart, rowStop : row\ boundaries$
$colStart, colStop : column\ boundaries$
$B : memory\ size \; */$

MAP$(M, rowStart, rowStop, colStart, colStop, B)$

1. if $((rowStart > rowStop)$ or $(colStart > colStop))$, then
   /* Match Table is consumed. */
   Return;
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$(M, rowStart, rowStop, colStart, colStop, B)$;

Figure 5: MAP Search algorithm.

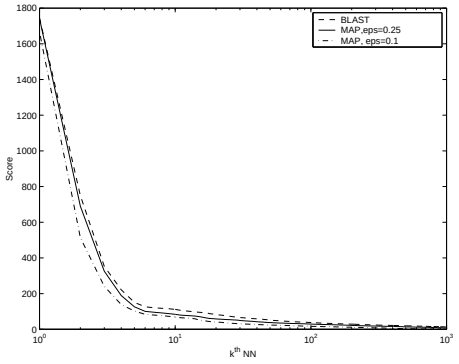


Figure 6: The average score for the first 1000 results for BLAST and MAP for different values of ϵ, for $|q|$ = 4000 and $\Delta = w =$ maximum possible resolution.

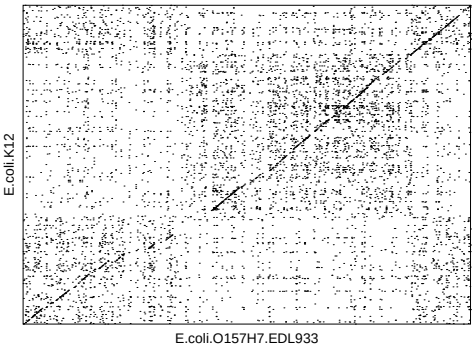


Figure 7: Match Table created by MAP for aligning E.coli.K12 with E.coli.O157H7.EDL933. The points correspond to marked entries of the Match Table.

Figure 7 shows a part of the match table constructed for aligning E.coli.K12 (4.6 M base pairs) with E.coli.O157H7.EDL933 (5.5 M base pairs). The diagonal run of marked entries depicts the similarity patterns. This figure shows that MAP can visualize similar patterns in a coarse grain without aligning the sequences. MAP created this matrix in less than 4 seconds.

4.2 Performance comparison

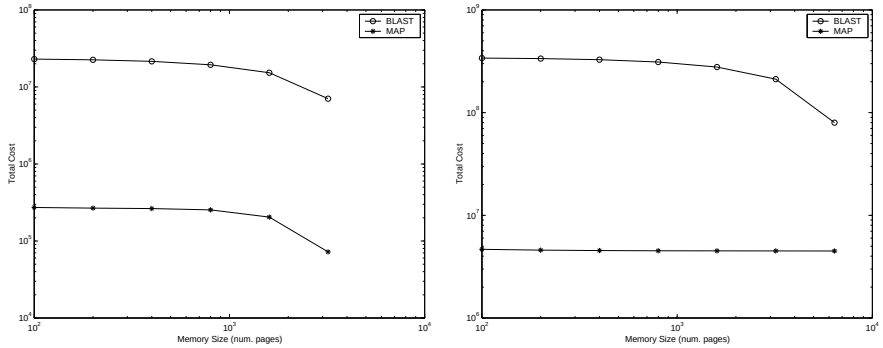
Our second set of experiments compares the performance of MAP to BLAST. Figure 8(a) shows the total cost of BLAST and MAP for aligning human chromosome 18 with mouse chromosome 18 for memory sizes 100, 200, 400, 800, 1600, and 3200 pages. In this experiment, for BLAST's advantage, we set BLAST to construct the hash table on the shorter string (i.e. mouse chromosome 18). For the considered memory sizes, MAP performed 74 to 97 times faster than BLAST.

Figure 8(b) shows the total cost of BLAST and MAP for the comparison of human chromosome 18 with human chromosome 21 for memory sizes 100, 200, 400, 800, 1600, 3200, and 6400 pages. MAP performed 20 to 74 times faster than BLAST for these datasets.

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 97 times faster than BLAST. 3) MAP works well even for small memory sizes.

5 Discussion

In this paper, we considered the problem of aligning huge genome strings. We also made significant improvements to the MRS index structure by presenting an algorithm that computes scores with affine gaps in the frequency domain. We presented an algorithm



(a) (b)

Figure 8: The total cost of BLAST and MAP in terms of page transfers for different memory sizes for aligning human chromosome 18 with (a) mouse chromosome 18, (b) human chromosome 21 when $\Delta = w$ = 4K.

that finds the local associations between a query string and a database string. The algorithm constructs a boolean *Match Table* which uses the MBRs of an index structure as its columns and substrings of the query as its rows. An entry in the Match Table is marked as *True* if the corresponding query substring and database MBR (which in turn represents a database substring) potentially contains a similar patterns. It is marked as *False* otherwise. The Match Table enables the user to visualize the similarity pattern between the strings prior to search in coarse grain.

The Match Table is split iteratively either vertically or horizontally minimizing the I/O and CPU costs. A hash table is constructed on the marked pages of the unsplit string. The marked pages of the split string are read sequentially and exactly matching substrings (i.e. seeds) of some certain prespecified size are found using this hash table. The seeds are then extended in both directions to find better matches. Later, the results are reported in a descending score order. The resulting search technique is called MAP. This MBR-based decomposition reduces the memory requirements and consequently the I/O cost. The CPU cost is also reduced since only the *True* entries of the Match Table need to be examined.

In our experiments, we used the BLAST[2] for comparison. According to our experimental results, MAP runs 20 to 97 times faster than BLAST. Furthermore, MAP can work well even for small memory sizes. Unlike the massive data structures used by current techniques, the size of the MRS index structure is only 1-2% of the database. MAP achieves these performance improvements while keeping 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.

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. We are currently building a web-based server which makes the MAP software available. We are also extending MAP to align protein strings using alphabet specific scoring schemes.

References

1. T. Kahveci and A. Singh. In *VLDB*, pages 351–360, Roma, Italy, September 2001.
2. S. Altschul and W. Gish. Basic local alignment search tool. *J. Molecular Biology*, 1990.
3. S.B. Needleman and C.D. Wunsch. *J. Molecular Biology*, 48:443–53, 1970.
4. T.F. Smith and M.S. Waterman. *J. of Molecular Biology*, March 1981.
5. E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, pages 251–266, 1986.
6. X. Huang and W. Miller. *Adv. Appl. Math*, 12:337–357, 1991.
7. S. Batzoglou, L. Pachter, J.P. Mesirov, B. Berger, and E.S. Lander. *Genome Research*, 10:950–958, 2000.
8. W.R. Pearson and D.J. Lipman. In *Proc. Natl. Acad. Sci*, volume 85, pages 2444–2448, 1988.
9. Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. *J. of Computational Biology*, 7(1-2):203–214, 2000.
10. T.A. Tatusova and T.L. Madden. *FEMS Microbiol. Lett.*, pages 247–250, 1999.
11. W. Gish. WU-blast. http://blast.wustl.edu.
12. D.J. States and P. Agarwal. In *ISMB*, 1996.
13. A. Califano and I. Rigoutsos. FLASH: Fast look-up algorithm for string homology. In *CVPR*, 1993.
14. S. Schwartz, Z. Zhang, K.A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller. *Genome Research*, 10(4):577–586, April 2000.
15. M. Ma, J. Tromp, and M. Li. *Bioinformatics*, 18(0):1–6, 2002.
16. D. Gusfield. Cambridge University Press, 1 edition, January 1997.
17. A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genome. *Nucleic Acids Research*, 27(11):2369–2376, 1999.
18. S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, 'E. Rivalsd, and M. Vingron. In *RECOMB*, Lyon, April 1999.
19. S. Kurtz and C. Schleiermacher. *Bioinformatics*, 15(5), 1999.
20. B. Seeger. *Information Systems*, 21(5):387–407, 1996.