Accelerating Substring Searching: Breaking the I/O Barrier

Tamer Kahveci Ambuj Singh
Department of Computer Science
University of California, Santa Barbara, CA 93106
{tamer,ambuj}@cs.ucsb.edu

Abstract

The exponential increase in the size of string databases makes substring search a challenging problem. Current techniques suffer from both disk I/O and computational cost because of extensive memory requirements and large candidate sets. We accelerate string search tools and reduce their memory requirements by precomputing the associations between the database strings and the query string. Our technique integrates the MRS index structure, proposed earlier for whole matching, and a memory resident substring searching tool such as BLAST. The associations computed between the query string and MBRs of the index structure are used to prune the database-query substring pairs that do not contain similar regions. The user can tune a cutoff parameter to decide on the amount of pruning and the quality of the result. The experimental results show that our technique prunes a significant amount of the database (typically 20-80%), thus reducing the disk I/O cost and the CPU cost significantly. The performance of BLAST increases by one order of magnitude without much of an impact on the output quality. In addition, the amount of memory need is 30-40 times less. Thus, this technique has the potential of making substring searching over large databases viable on ordinary desktop PCs.

1 Introduction

There is a frequent need to find similarities between strings. For example, the similarity of two DNA strings from different organisms may correspond to some functional or physical relationship between these organisms. Such similarities may be used to predict diseases, or to design new drugs. Significant breakthroughs have already been achieved in genome research using the analysis of similar genetic strings. Another application of substring searching is the identification of similar patterns in large text databases while allowing some amount of typographical errors. This application includes searching a word in a dictionary, or a phrase in a large collection of text. Spell checkers and web searchers are some specific examples of such applications. Video data can also be viewed as an event sequence if some prespecified set of events are detected and stored as a sequence. These events can be voices, faces, objects, or text. Video databases support a wide variety of applications including security cameras, interviews, documentaries, movies, and TV news. Similar event subsequences can be used to find related video segments.

The size of string databases has been growing exponentially in the past few years. The GenBank database, for example, has doubled every 15 months [7, 8]. With such a rapid growth, genome datasets and the associated access structures have become too large to fit in the main memory of a PC, leading to a large number of disk accesses and page faults (and therefore, slow response times) for homology searches.

That disk I/O is a current bottleneck can be seen from the following simple calculation. Consider a dataset that stores the entire human genome DNA sequence. The size of such a dataset will be approximately 1 Gb. A BLAST-like [1] index structure will utilize a hash table of more than 20 GB, far more than what can fit in the memory of even a powerful workstation. When such a hash table is stored on disk, we incur the cost of a random disk page read[1], roughly 10 ms, for every substring of seed length size in the query string. This implies a cost of at least $10n$ ms, n being the length of the query string, for finding the matching seeds in the input string. Assuming a similar cost for accessing the dataset in order to construct the HSPs (High Scoring Pair), the total cost of disk I/O becomes $20n$ ms. The final phase produces sequence alignments. Using

[1] Sequential disk reads can be more than 10 times faster than random disk reads.

```
A C T - - T A G C
  R   I I       D
A A T G A T A G -
```

Figure 1. Transformation of the string ACTTAGC to AATGATAG using edit operations.

a dynamic programming based scheme[2], two strings of length n can be aligned in roughly $2 \times 10^{-5}n^2$ ms on a 500 MHz PC. This implies that for query strings of length up to 10^6, the cost of homology searching is going to be dominated by disk access costs.

Most of the string search algorithms proposed so far are in-memory algorithms [4, 5, 17, 23, 30, 39, 42, 44]. That is, these techniques have to scan the whole database for each query. These techniques suffer from disk I/Os when the database is too large. In-memory algorithms can become impractical for string databases because the database size grows faster than the available memory capacity, and extensive memory requirements make the search techniques impractical. The number of CPU operations can be reduced by using index-based techniques [1, 3, 11, 13, 27, 28, 31, 40, 41, 45]. However, the typical size of these indexes is even larger than the size of the database. For example, the size of the SST-tree [15] is 200 times larger than the database size for the parameters specified in [15].

The distance between two strings is generally defined as the minimum number of edit operations to transform one string into the other. A string s_1 can be transformed into another string s_2 by using three *edit operations*, namely *insert*, *delete*, and *replace*, on individual characters of the string s_1. Figure 1 presents a transformation of the string ACTTAGC to AATGATAG using edit operations. The transformation given in Figure 1 consists of 1 replace, 2 insert, and 1 delete operations. Let m and n be the lengths of strings s_1 and s_2. The edit distance, $ED(s_1, s_2)$, and the corresponding edit operations can be determined in $O(mn)$ time and space using dynamic programming [17]. The space complexity can be reduced to $O(min\{m, n\})$ if only the edit distance is needed (i.e. the corresponding edit operations are not required). Some applications assign different weights to different edit operations or different character pairs [22], leading to a *weighted edit distance*. The time and space complexity of finding the weighted edit distance is also $O(mn)$ by using dynamic programming.

An *alignment* of strings s_1 and s_2 is obtained by matching each character of s_1 to a character in s_2 in increasing order. All the unmatched characters in both strings are matched with space. An alignment of strings ACTTAGC and AATGATAG is given in Figure 1. The *dashes* (i.e., -) in this figure correspond to spaces. Each character pair is assigned a score based on their similarity, and these values are stored in a *score matrix*. The value of an alignment is defined as the sum of the scores of all of their character pairs. *Global alignment* (or *whole matching*) of s_1 and s_2 is defined as the the maximum valued alignment of s_1 and s_2. Finding the similarity of two strings is the dual of finding the distance between them. *Whole-substring matching* of s_1 and s_2 is defined as the highest valued alignment of s_1 and all the substrings of s_2. *Local alignment* [39] (or *substring-substring matching*) of s_1 and s_2 is defined as the highest valued alignment of all the substrings of s_1 and s_2. Both global and local alignments can be determined in $O(mn)$ time using dynamic programming.

In a previous paper [26], Kahveci and Singh presented the MRS index structure and showed how it can be used for answering whole-substring queries. In this paper, we build on their ideas and consider substring-substring queries. Such queries are very important for biological applications where highly similar local matches matter more than global matches.

Our contribution is the development of a disk based substring-substring searching technique. Instead of developing all components from scratch, we reuse existing tools that are adept at memory-based searches. We show how these tools can be restructured to work in external memory with the help of an index structure. We use the *MRS index structure* [26] to estimate the similarity of the database substrings to the query substrings prior to actual search. Typical size of this index structure ranges between 1-2$\%$ of the database size.

Our algorithm constructs a boolean *Match Table* for a given query string and database string. An entry in the Match Table is marked as *True* if the corresponding query substring and database substring potentially contains 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.). Next, BLAST is run on the query and database substrings that are marked as *True*, reading the hash table and the database substrings. Later, the results are reported in a descending score order. We call this technique the *MAP* (MAtch table based Pruning) technique. Though our presentation and discussion incorporates BLAST, the general idea can be used to speed up and reduce the memory requirements of other search tools as well.

The experimental results show that, MAP runs 10 to 30 times faster than BLAST without decreasing the output quality.

[2]The results will be skewed further in favor of disk access costs for heuristic-based alignments that are used currently.

The maximum memory requirements of MAP is 30 to 40 times less than that of BLAST. This drastic reduction in CPU and I/O cost has the potential of making substring searches (in particular homology searches) viable on desktop PCs.

The rest of the paper is as follows. Section 2 discusses the related work. Section 3 discusses the MRS index structure, and presents an some generalizations. Section 4 explains the MAP technique. Section 5 discusses the space and time complexity of the MAP technique. Section 6 discusses the experimental results. We end with a brief discussion in Section 7.

2 Background

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 [32, 39]. This technique is a variation of the dynamic programming algorithm that finds the edit distance between two strings by generating a distance matrix of size $m \times n$. For long data and query strings, this technique becomes infeasible in terms of both time and space. Myers [30] improved the time and space complexity to $O(rn)$ by maintaining only the required part of the distance matrix. However, for large error rates r is $O(m)$, and hence the complexity is still $O(mn)$.

The SIM program [23] finds k best non-intersecting local alignments between two sequences using dynamic programming techniques. SIM guarantees to find optimal alignments. The space requirement of SIM is proportional to the sum of the input sequence lengths and the output alignment lengths.

FASTA [33] starts by finding exactly matching k-tuples between two strings (k is typically 6 for DNA strings and 2 for protein strings). In the second step, the score of the 10 best diagonal runs are determined by performing a gapless extension. Next, overlapping regions are combined, and the substrings whose score is less than a given threshold are discarded. Finally, each of the selected substrings are aligned using banded Smith-Waterman algorithm [39].

Altschul, Gish, Miller, Myers, and Lipman proposed BLAST [2] for finding local similarities. This program runs in two phases. In the first phase, all the substrings of the query of some prespecified length (typically between 3 and 11) are searched in the database for an exact match. In the second phase, all the matches obtained in the first phase are extended in both directions until the similarity between the two substrings falls below some threshold. This technique keeps a pointer to the starting locations of all possible substrings of the prespecified length in the database to speed up the first phase. Therefore, the space requirement of BLAST is more than the size of the database. BLS2SEQ [41] utilizes the BLAST algorithm for pairwise DNA-DNA or protein-protein sequence comparison. Its space requirements are similar.

MegaBlast [45] uses a greedy algorithm for aligning DNA sequences instead of traditional dynamic programming techniques. However, its memory requirements are similar to BLAST.

SENSEI [40] starts by finding exactly matching seeds of length 8. Later these seeds are extended by 8 base pairs at a time in both directions. SENSEI can work in both 2 bits per base and 1 bit per base modes. In the latter case, the bases are distinguished as purine or pyrimidine. The experimental results show that SENSEI is almost one order of magnitude faster than BLAST for evolutionary large distant sequences. Although the space requirement of SENSEI is less than BLAST, it is still larger than the database size since a pointer is kept to the starting location of all possible substrings of length 8.

QUASAR [11] is a good search tool for strongly similar strings. This technique maintains a suffix array to retrieve the locations of the q-grams. The database is split into blocks of a fixed size. A window is slid on the query sequence, and the number of common q-grams of the subquery specified by the window and each of the database blocks are counted using the suffix array. Later, database blocks which contain less common q-grams than a given threshold are eliminated, and BLAST is run on the remaining portions of the database to find the alignments. According to the experimental results in this paper, this technique is an order of magnitude faster than BLAST. This technique has three drawbacks: First, it has extensive memory requirements. The memory requirement is 9 times the database size at construction phase, and 5 times the database size at run time. Second, the performance deteriorates quickly if the compared sequences are not very close. Third, it has false drops since the substrings that span two consecutive blocks are not considered.

SST-tree [15] is a heuristics-based solution which runs in $O(logn)$ expected time. This technique splits the data strings into overlapping windows of length l for some prespecified overlap amount of Δ. For each such window, the technique counts the number of repetitions of all the possible *k-tuples*, and stores this value in a σ^k dimensional vector, where σ is the alphabet size. Later, these vectors are indexed using a hierarchical binary tree. The authors propose to approximate the similarity between the query string and a substring by using the L_1 norm of the difference between these vectors. Experimental results show that this technique runs 25 to 50 times faster than BLAST. There are two drawbacks with this method: it allows false drops, and the index size increases exponentially with k. For the parameters suggested in this paper ($k = 5, \Delta = 5$), the size of the index structure is 200 times larger than the size of the database, making this technique impractical for real applications.

Myers [31] proposed a technique similar to BLAST for the whole-substring matching problem. This technique assumes a lower bound l on the length of the query string ($l = log(n)$ here). All possible strings of length l are mapped to integers using a perfect hashing function. Later, the leftmost points of all the occurrences of these strings in s are stored in separate lists. For a given query q and query radius r, the technique generates the set of strings which are within edit distance of r to q, called *condensed r-neighborhood*. The strings in the condensed r-neighborhood are searched in the index to find the answers to the query. If the query length is larger than l, the technique splits the query string into subqueries, searches each subquery separately and combines the results. The author proves that, if the database is created as a result of equi-probable Bernoulli trials, then the technique runs in sublinear time. There are two drawbacks with this technique. First, although the space complexity is $O(n)$, the index size can be 7-9 times larger than the data size. This may cause a drop in performance if the index does not fits in memory. Second, the worst case running time complexity of this technique is very high.

A similar technique is proposed by Baeza-Yates and Navarro [3] for the whole-substring matching problem. The authors propose to keep a dictionary that stores all possible substrings of length l that occur in s. The dictionary also keeps an array for each substring to store the starting locations of that substring in s. For a given query q with range r, the technique splits q to $r + 1$ substrings. By the Dirichlet principle, at least one of these substrings exactly matches with an answer substring. The technique seeks exact matches for the subqueries in the first phase. Later, each matching substring is linearly expanded to check if it is really an answer. The authors show that in most practical applications the expected run time is sublinear. This technique suffers from space problems for large l, m, and r. Furthermore, the experimental results presented show that this technique runs slower than the online method for large r.

Pevzner and Waterman [34] considered a similar filtration technique for whole-substring matching problem. The authors observed that if the edit distance between two strings of length n is less than r, then they must contain at least $n - k + 1 - kr$ common k-tuples. This technique also suffers from space problems since additional space is required to maintain k-tuples.

Muthukrishnan and Sahinalp [29] proposed an index structure for approximate nearest neighbor search. This technique uses an index structure based on suffix arrays and a partitioning of the pattern. The resulting index structure is four times the size of the database.

MUMmer [13] uses suffix trees to find the maximal unique matches. Later these matches are sorted and the gaps between close matches are filled with inserts and deletes. This technique suffers from extensive memory requirements due to use of suffix trees. Size of the suffix trees can be reduced by removing the suffix links [24]. However, its size is still 68 times larger than the database size after this removal [24].

REPuter [27] finds the repeating patterns in genetic strings by constructing a suffix tree on the fly. This technique suffers from extensive memory requirements since it maintains a suffix tree.

Baeza-Yates and Perleberg [5] consider a restricted version of the problem in which only *replace* operation is allowed to transform the strings. This is the same as searching the query string q in a string s with at most r mismatches. They present two algorithms in this paper. The first algorithm stores the locations (i.e. offsets) of each character of the alphabet in q in different lists. A separate counter, initialized to 0, is maintained for each character of s. Later, the data string s is scanned. For each character in s, the possible offsets in q are obtained, and the corresponding counters are incremented by 1. Once s is scanned, the values of the counters represents the similarity of the substrings starting at that location to q. The worst case running time of the algorithm is $O(n + A)$, where A is the size of the answer set. The space requirement is $O(m + n)$. The second algorithm splits q into $r + 1$ substrings, so that at least one of the substrings exactly matches with a substring in s. For each exact match, they extend it to check if it really corresponds to an answer. The techniques presented here works well only for very small values of r.

A special case of the substring matching problem is *exact matching* (i.e. $r = 0$). One can solve this problem using suffix trees [17] in which all the suffixes of a database string are stored in a tree. However, the size of the suffix tree may be more than ten times larger than the database size. Manber and Myers [28] propose a data structure called *suffix arrays* to reduce the space requirement for the index structure. However, the space requirement is still more than four times the database size. Ferragina and Manzini [14] proposed a technique to compress the suffix arrays, decreasing the query performance slightly.

Jagadish, Koudas, and Srivastava consider the problem of exact matching with wild-card for multidimensional strings in [25]. The authors map all the strings to real space based on their lexical order. Later, these multidimensional points are indexed using R-trees [18]. The strings within a data page are stored using *elided tries* for each dimension. This technique works efficiently if the database consists of short strings. However, the performance degrades for long strings. This is because it becomes more difficult to distinguish long strings by inspecting (the much shorter) prefixes of these strings.

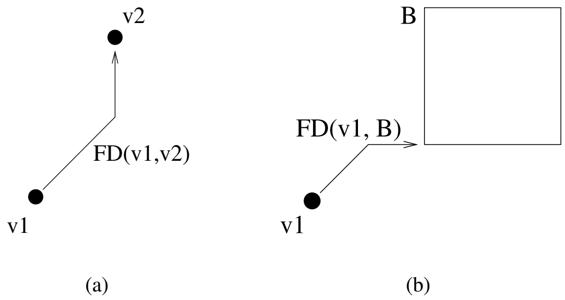

Figure 2. An illustration of frequency distance when $|\Sigma| = 2$. (a) Frequency distance between two points. (b) Frequency distance between a point and an MBR

Gravano, Ipeirotis, Jagadish, Koudas, Muthukrishnan, and Srivastava [16] use three pruning techniques to perform approximate string joins faster: Length Filtering, Count Filtering, and Position Filtering. Their technique maintains a table for positional q-grams of the database. This technique is suited more for whole-substring matching. The size of the q-gram table can also become a limiting factor for large databases.

3 Improving MRS index structure

Section 3.1 introduces the MRS index structure [26]. Section 3.2 discusses a new measure of frequency distance. Section 3.3 extends the indexing technique to scores and affine gaps. Section 3.4 considers alphabet-specific scoring scheme.

3.1 Background on the 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=TACTTAG is a DNA string, then $f(s) = [2, 1, 1, 3]$. 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 $f(s)$. An illustration of the frequency distance between two frequency vectors is given in Figure 2(a). $FD(v_1, v_2)$ is equivalent to the length of the shortest path between v_1 and v_2, such that at each step on this path either the value of a dimension changes (increases or decreases) by one or the value of one dimension increases by one while that of another dimension decreases by one.

The frequency distance between the frequency vectors of two strings is a lower bound on their edit distance. This is illustrated in Figure 3. 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. The frequency distance for DNA strings can be computed in constant time using 8 integer additions and 4 integer comparisons (see [26]).

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

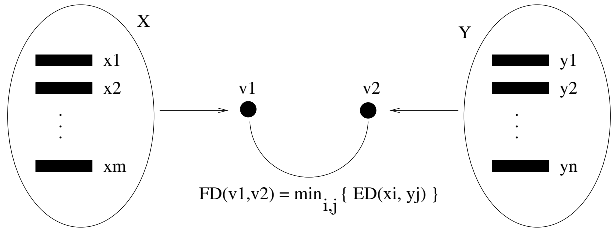

Figure 3. Two frequency vectors v_1 and v_2, the set of strings corresponding to these frequency vectors, and the frequency distance between v_1 and v_2 in terms of edit distance.

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.

$T_{i,j}$ 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 stores frequencies at different resolutions, this index structure is called as the *Multi Resolution String (MRS) Index Structure* [26]. 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.

Note that one may consider using R-tree [6, 18] or other similar multidimensional indexing techniques [9, 10, 12, 19, 35, 38, 43] instead of MRS index structure to index frequency vectors. These index structure have two major drawbacks: First, frequency vectors and pointers to their corresponding substring must be stored for each leaf level entry. As a result of this, the size of the index structure increases dramatically. Second, postprocessing incurs a high amount of random disk I/O since the frequency vectors in an MBR can correspond to database substrings in any order.

Analogous to the frequency distance between two frequency vectors, the frequency distance between a frequency vector v_1 and an MBR B, $FD(v_1, B)$, is defined as the length of the shortest path between v_1 and B. This is illustrated in Figure 2(b). Since $FD(v_1, B) \leq FD(v_1, v_2)$ for all v_2 contained in B, we conclude that if $FD(v_1, B)$ is greater than r then the edit distance between a string q (for which $v_1 = f(q)$) and all the strings s (for which $f(s) \in B$) is greater than r.

3.2 A new frequency distance

Let $S = \{f(s) | s \in \Sigma^*\}$ be the set of all possible frequency vectors of the strings from alphabet Σ, where $|\Sigma| = \sigma$. Set S lies in the non-negative quadrant of the σ-dimensional integer space. Define a subset $S_w = \{f(s) | s \in \Sigma^* \text{ and } |s| = w\}$, where w is a positive integer. Set S_w defines all possible frequency vectors of strings of length w. Lemma 1 explains the relation between S and S_w.

Lemma 1 *S_w is the intersection of S with the (σ-1)-dimensional plane which passes through $(w, 0, 0, ..., 0)$ and is perpendicular to the vector $(1, 1, ..., 1)$.*

Proof:
Let $v \in S_w$, then $\sum_{i=1}^{\sigma} v[i] = w$. Hence, each frequency vector $v \in S_w$ constitutes a solution to the equation $x_1 + x_2 + ... + x_\sigma = w$. The set of solutions to this equation is the (σ-1)-dimensional plane explained in this lemma. □

An example of S and S_w is presented in Figure 4. The shaded plane represents S_w. Since S_w contains the frequency vectors of all the strings of length w, we call it the *Data Plane*. The same figure also contains an MBR, B, from an MRS index structure for resolution w. Since B contains frequency vectors of substrings of length w, the frequency vectors that constitute B lie on the S_w plane. Therefore, only the intersection of B with S_w is sufficient to represent the frequency vectors that form B. Using this fact, Lemma 2 constructs a new lower bounding distance function which is never less than the frequency distance to an MBR.

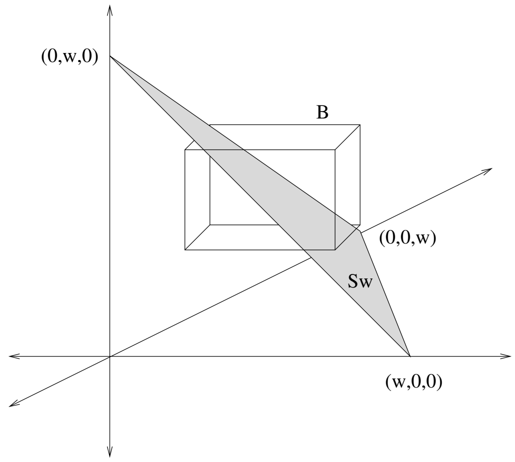

Figure 4. The frequency vector space S, a subspace S_w, and an MBR B in S when $|\Sigma| = 3$.

Lemma 2 *Let B be an MBR of an MRS index structure for resolution w. Let q be a string. Let s be a string of length w such that $f(s) \in B$, then*

1. $FD(f(q), B) \leq FD(f(q), B \cap S_w)$.
2. $FD(f(q), B \cap S_w) \leq EditDistance(q, s)$.

Based on Lemma 2, $FD(f(q), B \cap S_w)$ is a better lower bounding distance function than $FD(f(q), B)$ (i.e., it potentially prunes larger amount of the database). The idea of using S_w to constrain the distance computations is a new one and an improvement over the notation of distance as originally proposed for the MRS index structure [26]. For simplicity, we will write $FD_w(f(q), B)$ instead of $FD(f(q), B \cap S_w)$.

$FD_w(f(q), B)$ can be computed efficiently using w, and lower and higher coordinates of B. For example, let the lower and higher coordinates of B be respectively $L = [20, 20, 20]$ and $H = [50, 50, 50]$ for $w = 100$. Let $f(q) = [55, 15, 40]$. Let $x = [x_1, x_2, x_3] \in B \cap S_w$ be the point that minimizes $FD(f(q), x)$. Since, $20 \leq x_1 \leq 50$ choose $x_1 = 50$ (i.e. the closest point in B to the first entry of $f(q)$), similarly choose $x_2 = 20$ and $x_3 = 40$. These values minimize $FD(f(q), x)$ since only 5 increment/decrement operations are needed to transform $f(q)$ to x. On the other hand $x_1 + x_2 + x_3 = 110$ is greater than w. Therefore we have to decrease the sum of entries of x by 10. We can do this only by decreasing x_1 or x_3 (x_2 is already the minimum possible value). As a result, the sum of the values of x_1 and x_3 becomes 10 less than the corresponding entries of $f(q)$, and hence $FD_w(f(q), B) = 15$.

Figure 5 presents an algorithm that computes the frequency distance $FD_w(v, B)$ between a frequency vector v and an MBR B in σ dimensions. The algorithm starts by initializing the increment and decrement amounts and the sum to zero (Step 1). Later, the minimum increment or decrement amount is computed and the value of the closest point to the query vector is added cumulatively for each dimension (Step 2). The minimum amount of edit operations needed to translate the closest point found in Step 2 to the S_w plane is added to the increment or decrement amounts (Steps 3 and 4). In the worst case, $3 \times (\sigma + 1)$ integer additions and $\sigma + 1$ integer comparisons are sufficient to compute $FD_w(v, B)$. For DNA strings this number is 15 integer additions and 5 integer comparisons.

3.3 Extending the technique to scores and affine gaps

The dual problem of finding the edit distance between two strings is to find the score of the best alignment of these two strings. A positive score (S_{match}) is assigned to the pair of characters that match and a penalty ($S_{mismatch}$) is assigned if they mismatch. The simpler version of this problem assigns a fixed penalty (S_{gap}) to all the gaps. The problem of finding the minimum number of edit operations to transform one string to another reduces to finding the highest scoring alignment when $S_{match} = 1, S_{mismatch} = S_{gap} = -1$.

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

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. if $sum < w$ then
   dec += $w - sum$
4. else if $w < sum$ then
   inc += $sum - w$
5. return $max\{inc, dec\}$

Figure 5. Computation of FD_w

For biological strings, the likelihood of having two gaps of at different locations is less than that of having two consecutive gaps. This is because if a nucleotide is cut from a DNA string during a mutation, then it is likely that a longer piece of that DNA string is cut during that mutation. This is captured in the current search tools by using the *affine gap model*. In this model, the gap penalty, S_{gap}, is replaced by a (potentially large) gap open, S_{gap_open}, and a (smaller) gap extend, S_{gap_extend}, penalty. A gap of length g incurs a penalty of S_{gap_open} for the first entry in the gap and S_{gap_extend} for each of the remaining entries, adding up to $S_{gap\_open} + (g-1) \times S_{gap\_extend}$. The default values in BLAST for nucleotides is $S_{match} = 1$, $S_{mismatch} = -3$, $S_{gap\_open} = -5$, and $S_{gap\_extend} = -1$. The score of the alignment in Figure 1 using these default values is $5 \times 1 + 1 \times -3 + 2 \times -5 + 1 \times -1 = -9$. Since the gap open penalty is larger than the others, usually BLAST outputs contain only a few 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 using the frequency distances. Consider the example in Section 3.2. The shortest path that translates $f(q) = [55, 15, 40]$ into $B \cap S_{100}$, where $B.L = [20, 20, 20]$ and $B.H = [50, 50, 50]$, consists of 5 increment/decrement operations (i.e. modify operation), and 10 decrement operations (i.e. delete). The former operations correspond to mismatches and the latter correspond to gaps. Since, there is no information about the number of gaps, we assume a single gap of length 10 to minimize total penalty. Note that one can replace a modify operation with a delete followed by an insert, but we rule this out to minimize the number of gaps. Since $w = 100$, the number of matches is not greater than 95 (i.e. $100 - 5$). As a result, we conclude that the score of the best alignment of q to a string contained in B can not be greater than $95 \times S_{match} + 5 \times S_{mismatch} + S_{gap\_open} + 9 \times S_{gap\_extend}$ = 66.

Figure 6 presents an algorithm, $FS_w(v, B)$, to compute an upper bound to the score of the best alignment of a substring whose frequency vector is v to a string in B for window size w. The algorithm starts by finding the number of mismatches similar to $FD_w(v, B)$ algorithm in Figure 5 (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). In the worst case, $3 \times \sigma + 7$ integer additions, 5 integer multiplications and $\sigma + 1$ integer comparisons are sufficient to compute $FS_w(v, B)$. For DNA strings this number is 19 integer additions, 5 integer multiplications, and 5 integer comparisons.

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

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. $ScoreInc = (min\{sum, w\} - inc) \times S_{match} + inc \times S_{mismatch}$
4. $ScoreDec = (min\{sum, w\} - dec) \times S_{match} + dec \times S_{mismatch}$
5. if $w < sum$ then
   $ScoreInc$ += $S_{gapopen} \times (sum - w - 1) + S_{gap\_extend}$
6. else if $sum < w$ then
   $ScoreDec$ += $S_{gapopen} \times (w - sum - 1) + S_{gap\_extend}$
7. return $min\{ScoreInc, ScoreDec\}$

Figure 6. Computation of FS_w

3.4 Extending to alphabet specific scores

Genome search tools for proteins (e.g. BLASTP) usually employ an alphabet specific scoring model. In this model, each character pair is assigned a score. This is represented using a *score matrix*. Some of the popular score matrices are PAM and BLOSUM [20, 21, 22]. The algorithm in Figure 6 can be easily extended to find an upper bound to the best score under this model. In order to do this, once the amount of increments and decrements for each character are determined in Step 2, the algorithm iteratively matches an increasing character and a decreasing character that maximizes the score. This increases the number of integer additions only by the size of the alphabet.

4 MAP algorithm

Now we are ready to discuss how our technique can be confined with an in-memory searching tool to produce an efficient disk-based search tool. We employ BLAST as our example of the in-memory search tool because of a number of reasons. First, BLAST is highly optimized to work efficiently. Second, the notion of distance for homology searching is complex, meaning that other kinds of substring searching can only be simpler. Finally, the impact of lowering the cost of searching will have a huge impact in the area of bioinformatics as well as databases.

Section 4.1 discusses the space requirement of BLAST. Section 4.2 presents our improved MAP technique. Section 4.3 shows how the improvements can be extended to other search tools.

4.1 Analysis of the space requirement of BLAST

Current index based techniques have two major drawbacks. First, the sizes of the index structures are very large. Some of them are even more than 200 times larger than the database size. Therefore, they require excessive amount of memory for both the index structure and the candidate set. As a result, they suffer from disk I/Os even in the index search phase. Second, since most of the techniques begin by finding a short seed of a fixed length (e.g. For BLAST, the seed length is 11 for nucleotides and 3 for proteins.), the size of the candidate set can be as large as $O(D|q|)$, where D is the database size and $|q|$ is the query size. We prove this fact in Lemma 3.

Lemma 3 *Let x and y be two strings from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$, then the expected number of common substring pairs $<x_i, y_j>$, where $|x_i| = |y_j| = L$, and x_i is a substring of x, and y_j is a substring of y is*

$$\frac{(|x|-L+1).(|y|-L+1)}{\sigma^L}$$

if all the letters in the alphabet occur with the same probability.

Proof:
Let x_i be any substring of x, where $|x_i| = L$. Since there are σ^L possible strings of length L, the probability that a randomly generated string y_j of length L is equivalent to x_i is $\frac{1}{\sigma^L}$. Since y has $|y| - L + 1$ substrings of length L, it contains approximately $\frac{|y|-L+1}{\sigma^L}$ substrings equivalent to x_i. Since x contains $|x| - L + 1$ substrings of length L, the expected number of common substring pairs is $\frac{(|x|-L+1).(|y|-L+1)}{\sigma^L}$. □

As a consequence of Lemma 3, the candidate set becomes very large for long queries and large databases. For BLAST nucleotide search, the expected number of the seeds is approximately $\frac{|x|\times|y|}{4^{11}}$. Once the initial seeds are generated, BLAST extends all the matches obtained in the first phase in both directions until the similarity between the two substrings falls below some threshold. In order to find the locations of the seeds quickly, BLAST keeps a pointer to the starting locations of all possible substrings of the prespecified length in the database using a hash table. Lemma 4 computes the expected number of entries of the hash table that needs to be examined if the size of the initial seed set is known.

Lemma 4 *Let A be the number of possible different values for the seeds, then n randomly selected seeds correspond to $\frac{A.n}{A+n-1}$ different seeds on the average.*

Proof:
Let $p_{k,n}$ be the probability that n seeds span exactly k different seeds, then

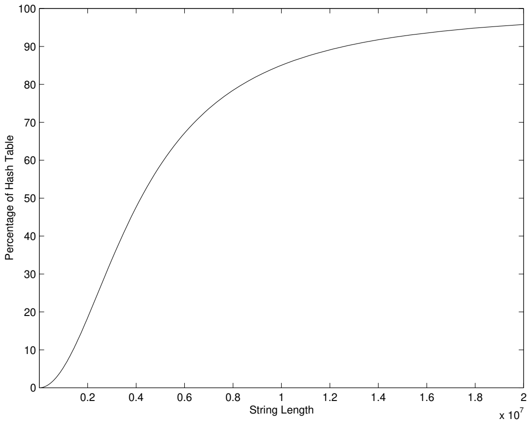

Figure 7. Percentage of hash table accessed by BLAST for increasing length of compared strings.

$$p_{k,n} = \frac{C_k^A . C_{k-1}^{n-1}}{C_{A-1}^{A+n-1}}.$$

The expected number of different seeds is

$$\begin{aligned} E[A, n] &= \sum_{k=1}^{min\{A,n\}} k.p_k \\ &= \sum \frac{k.C_k^A . C_{k-1}^{n-1}}{C_{A-1}^{A+n-1}} \\ &= A. \sum \frac{C_{A-k}^{A-1} . C_{k-1}^{n-1}}{C_{A-1}^{A+n-1}} \\ &= \frac{A.C_{A-1}^{A+n-2}}{C_{A-1}^{A+n-1}} \\ &= \frac{A.n}{A+n-1}. \qquad \square \end{aligned}$$

Using Lemmas 3 and 4, we can conclude that the expected number of different seeds for BLAST ($A = 4^{11}$) is approximately $\frac{4^{11}.|x|.|y|}{4^{22}+|x|.|y|}$ for comparing two DNA strings x and y. Figure 7 plots the percentage of the hash table that needs to be accessed by BLAST for an increasing length of compared strings. As seen from the figure, BLAST inspects 85% of the hash table when the length of the strings compared is 10^7. The length of queries against human chromosomes can be much larger than 10^7. Therefore, BLAST accesses almost the entire hash table for such strings. This implies that a large number of pointers to disk are traversed, leading to significant I/O.

4.2 High-level description of the MAP technique

Unlike the indexes of the genome search tools discussed above, the size of the MRS index structure is about 1-2$\%$ of the database size if the box capacity is 1000. Furthermore, the index searching is extremely fast since only 19 integer additions and 5 integer comparisons are needed per MBR (at most). Therefore, the MRS index structure can be embedded into current search techniques with a negligible computation and space cost. We propose to increase the performance of BLAST by estimating locally similar substrings of the database to a given query. The resulting technique is called *MAP*. Figure 8 illustrates the different steps of this technique.

The user submits a query along with two search parameters: quality cutoff and shift amount. These parameters affect the sensitivity and the performance of search and are explained later. A *Match Table* as shown in Figure 9 is then constructed by considering locally similar substrings of query and database strings. For each locally similar pair, BLAST (or some other search engine) is invoked to compute locally similar substrings and their scores. Finally, the answer set is reported in a decreasing score order. We first detail the construction of the match table.

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 MRS index structure. As shown in Figure 9, 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

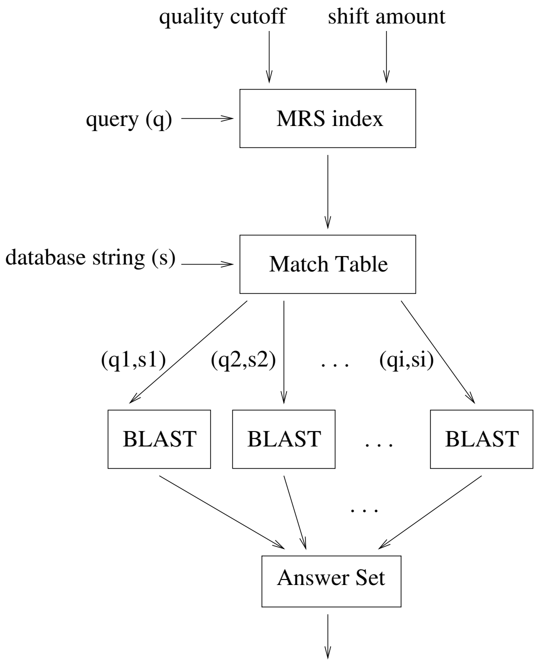

Figure 8. Control flow within MAP.

q \ s	s1	s2	s3	s4	s5
q1	T				
q2	T		T	T	
q3			T		
q4				T	

Figure 9. An illustration of the match table constructed on q and s using the MRS index structure. An entry is set to T (i.e. *True*) if the corresponding substrings potentially contain similar patterns. Unmarked entries are *False*.

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 s, the second column corresponds to the second 1000 substrings, and so on. A number of subqueries, $q_1, q_2, ..., q_{n_w}$, are constructed by sliding a window of length w on q with a shift amount of Δ (We will elaborate on the choice of w and Δ later.). Each such query generates a row of M. In the figure, four subqueries q_1, q_2, q_3, q_4 have been shown. Entry $M_{i,j}$ is set to *True* if the MBR B_j potentially contains a similar substring to the query substring q_i.

$/*\, INPUT \quad q : query\, sequence$
$\qquad\qquad \epsilon : error\, rate$
$\qquad\qquad \Delta : shift\, amount$
$OUTPUT \quad M : match\, table \quad */$

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

Figure 10. Construction of Match Table.

Figure 10 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 finding the largest resolution in the MRS index structure that is less than $|q|$ (Step 1). A *cutoff threshold*, τ is determined using this resolution and the error rate (Step 2). A window of the specified resolution is than 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. The default value for ϵ is 0.25 and for Δ it is w. The time and space complexities of *FIND_MATCHES* algorithm are $O(\frac{|q|-w}{\Delta}.\frac{|s|-w}{c})$, where c is the box capacity of MBRs and $|s|$ is the total database size. High values of c ($\approx$ 1000) make this negligible compared to other costs.

Once the match table is constructed, BLAST is called to search the query substrings within database substrings that are marked as *True*. For example, since $M_{1,1}$ is marked (in Figure 9), q_1 must be searched within B_1. In order to do this, BLAST needs to find the seeds within B_1 corresponding to q_1. Other BLAST searches that need to be carried out are q_2 within B_1, q_2 and q_3 within B_3, and q_2 and q_4 within B_4. The second and the last columns of the match table are unmarked (i.e. *False* columns). Therefore, BLAST does not need to search these parts of database nor store the hash table entries corresponding to these parts in memory.

We iterate through the database strings on an MBR by MBR basis. The database substrings contained in MBR B_j are accessed only if the match table, M, contains at least one *True* entry in column j. The database strings corresponding to such an MBR B_j are read using an optimal disk read schedule [36, 37]. 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 each MBR, we construct the initial seed set by considering the entries in the corresponding column that are set to *True*. Consecutive subqueries in an MBR may have overlapping substrings if $\Delta < w$. In order to avoid extending the same seed multiple times due to overlapping regions of consecutive query substrings, we merge consecutive substrings if both of them are marked *True*. For example, in Figure 9, $M_{1,1}$ and $M_{2,1}$ are marked. Therefore, we combine q_1 and q_2 to search within

B_1 instead of searching q_1 and q_2 separately.

The MRS index structure can be easily embedded on top of BLAST without changing its internal structure. A call to the subroutine that computes the Match Table is performed prior to BLAST. Later, BLAST is run once per MBR by restricting the seed set to the associations determined by the Match Table.

4.3 Extending the MAP technique to other genome search tools

Our general technique is to (1) partition a database of strings over alphabet Σ into sets of substrings, (2) map the information of each substring into integer space with the help of frequency vectors, (3) coalesce the information about each set with the help of an MBR, (4) use the MBRs and frequency distance to compute associations (match table) between query substrings and MBRs, and (5) iteratively search through the database strings one MBR after another for the likely matches. The general approach can be extended readily to other string search tools: steps 1-4 remain unchanged, only step 5 needs to be specialized. Below we discuss these specializations for a few string search tools.

- *FASTA*: Initial exact matches of FASTA are generated only on the *True* entries of the Match Table. Since FASTA starts with finding exact matches and considers only gapless extensions, we believe that our technique will give high quality results even for very small cutoff values.
- *SENSEI, BLS2SEQ, WU-BLAST, MegaBLAST, PowerBLAST*: These programs generate an initial seed set similar to BLAST. The main differences between these tools and BLAST are that the length of the initial seeds are different, and they use different techniques to extend the initial seeds. Therefore, our technique can be embedded into these tools simply by restricting initial seed set to the *True* entries of the Match Table. Since these techniques are very similar to BLAST in spirit, they will have similar speedups as BLAST.
- *SIM*: We can call SIM only on the *True* entries of the Match Table. Since this is a dynamic programming technique, we expect better speedups for this technique.
- *MUMmer*: The suffix tree of MUMmer can be constrained to the *True* entries of the Match Table. Since the size of the suffix trees increases rapidly, we expect dramatic reduction in the space requirement of MUMmer if it is used along with the MRS index structure.

5 Performance analysis of MAP

We analyze the performance of MAP relative to the performance of BLAST. In order to carry out this analysis, let
p_c = ratio of columns with at least one *True* entry to the total number of columns.
p_e = ratio of the *True* entries to the total number of entries.

5.1 Space analysis

The MAP technique stores the query, the MRS index structure, and the Match Table in memory at all times. Their combined space requirement is 1-2% of the database size. The rest of the memory is utilized by the BLAST search tool: a) Storage of initial seeds that are extended to HSPs (High Scoring Pairs), b) Storage of hash table entries that point to the database substrings on disk. This is illustrated in Figure 11. Let
S_{seed} = space requirement of BLAST for storing the initial seeds.
S_{hash} = space requirement of BLAST for storing the hash table.

Since MAP processes columns (MBRs) separately, it only needs to store the hash table and seeds corresponding to one column at a time. Once the part of the database corresponding to a column of the Match Table is searched, the current hash table and HSPs are removed from memory, and the hash table and the HSPs for the next column are read into memory. Therefore, if the Match Table contains M_c columns, then the maximum memory requirement of MAP for any point of time is approximately equal to $\frac{S_{seed}+S_{hash}}{M_c}$.

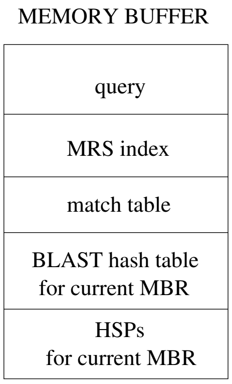

Figure 11. Memory usage of MAP.

5.2 I/O cost analysis

BLAST incurs I/O cost for two reasons. First, the database strings are read from disk and second, the hash table is read from disk. Let t_1 denote the former cost and t_2 denote the latter cost. MAP reads the database substrings only for those columns that have at least one *True* entry. This implies that it needs to read a factor of p_c less number of database substrings. A similar argument implies that the size of the hash table entries accessed by MAP is also reduced by a factor of p_c. The total I/O cost of the MAP technique is thus between $p_c.(t_1 + t_2)$ and $t_1 + t_2$, the actual value depending on the optimal disk scheduling [36, 37].

5.3 CPU cost analysis

The MAP technique first uses the MRS index structure to generate the Match Table. As mentioned in Section 4.2 this cost is $O(\frac{|q|-w}{\Delta}.\frac{|s|-w}{c})$, where w is the window size, Δ is the shift amount, and c is the box capacity. Since c is quite high ($\approx$ 1000) and the frequency distance between a query and an MBR can be computed very efficiently (at most 19 integer additions and 5 integer comparisons), this overhead is negligible with respect to other costs.

BLASTs major CPU cost is on account of extension of initial seeds to HSPs. Let t_{CPU} denote this cost. MAP needs to compute this extension only for the seeds remaining after the initial pruning. This implies that the CPU overhead of MAP is $p_e.t_{CPU}$.

6 Experimental results

For our experimental evaluation, we used chromosome 18 (*chr18*) from *homo sapiens* database. We downloaded the source code of the BLAST program, and implemented the MRS index structure for window sizes w = {256, 512, 1024, 2048, 4096}, and box capacity $c = 1000$. Later, we implemented MAP by embedding the MRS index structure as a preprocessing step to BLAST. We generated five query sets from, chr-18 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 chr18 dataset using BLAST and MAP with quality cutoff values of $\epsilon = \{0.1, 0.2, 0.25\}$ on a 400 MHz Pentium II computer with 1 Gb memory.

6.1 Quality comparison

Our first experiment set compares the quality of outputs. Figure 12 plots the score of the first 1000 outputs of BLAST and MAP for $\epsilon = \{0.1, 0.25\}$ for $|q| = 8000$, when w is chosen as the maximum possible resolution in the index structure and $\Delta = w$. The results for $\epsilon = 0.2$ are not shown, because they were very close to those of $\epsilon = 0.25$. The results for other query lengths were similar to these figures. The quality of MAP outputs are close to those of BLAST even when $\epsilon = 0.1$.

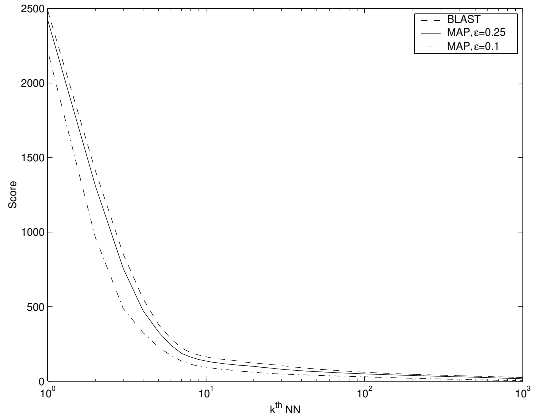

Figure 12. The average score for the first 1000 results for BLAST and MAP for different values of ϵ, for $|q|$ = 8000 and $\Delta = w =$ maximum possible resolution.

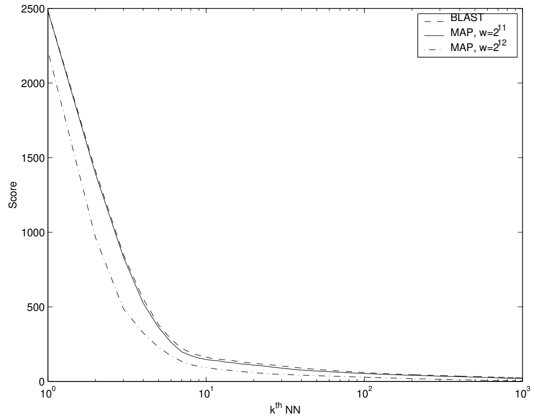

Figure 13. The average score for the first 1000 results for BLAST and MAP for different values of w, for $|q|$ = 8000, $\Delta = w$, and $\epsilon = 0.1$.

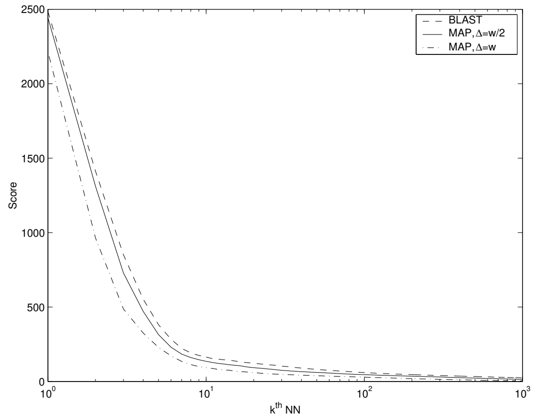

Figure 14. The average score for the first 1000 results for BLAST and MAP for different values of Δ, for $|q|$ = 8000, w = maximum possible resolution, and $\epsilon = 0.1$.

Δ	512	1024	2048	4096
MAP ($w = 2K$)	143.6	141.3	133.1	NA
MAP ($w = 4K$)	137.7	133.0	119.6	85.7

Table 1. The average score of the first 100 results for MAP for different values of Δ, for w = 2K and w = 4K, when $\epsilon = 0.1$.

The deviation between BLAST outputs and MAP outputs are about 5% for high scoring outputs, and 20-30% 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. The deviation between BLAST and MAP is on account of two reasons. 1) Since MAP considers a single resolution for a query sequence, it can miss similar short substrings. This effect is observed for the lower scoring results. 2) Substrings that are split across two consecutive query substrings are not considered. This effect leads to the deviation in the high scoring results.

One way to improve the quality of MAP outputs is to choose a smaller window size (this reduces the amount of error for high scoring region as well as low scoring region as explained earlier). Figure 13 shows the score of the first 1000 outputs of BLAST and MAP for $w = \{2^{11}, 2^{12}\}$ when $\epsilon = 0.1$, $|q| = 8000$, and $\Delta = w$. The results for other values of $|q|$ are similar. MAP almost overlaps with BLAST when $w = 2^{11}$. The difference between the curves for $w = 2^{11}$ and $w = 2^{12}$ shows the improvement in score by choosing a smaller window size (i.e. finer grain). For smaller values of w (not shown in the figure), the results are even better.

Another way to improve the quality of MAP outputs is to choose a smaller value of Δ. This will reduce the number of substrings that are not considered on account of lying across two consecutive queries in the high scoring region. Figure 14 presents the score of the first 1000 outputs of BLAST and MAP for $\Delta = \{w/2, w\}$, when $\epsilon = 0.1$, $|q| = 8000$, and $w = 2^{12}$. The deviation of MAP reduces to 1-5% by reducing the shift amount to $w/2$. The results for smaller values of Δ are not shown since they are indistinguishable from those of BLAST.

Table 1 presents the average score of the first 100 outputs for MAP for various values of w and Δ, for $|q| = 8000$, and $\epsilon = 0.1$. According to this figure, the quality of MAP improves as w and Δ decrease.

6.2 Memory requirements of MAP and BLAST

Our next experiment compares the memory requirements of the MAP and BLAST techniques. The memory requirement is computed as the sum of the sizes of the hash table needed, the database strings processed, and the seed set. Figure 15 plots the memory requirements for different query lengths when w is chosen as the largest possible value and $\Delta = w$ (i.e. highest possible pruning) in log scale. We did not plot the results when $\epsilon = 0.1$ and $\epsilon = 0.2$, because they were indistinguishable from the plot for $\epsilon = 0.25$ in this figure. For all values of $|q|$ and w, MAP needs much less memory than BLAST. For

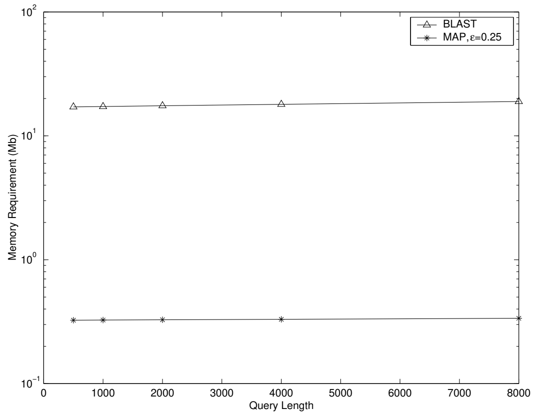

Figure 15. Memory requirements of BLAST and MAP for different values of $|q|$, when $\Delta = w$ = maximum possible resolution.

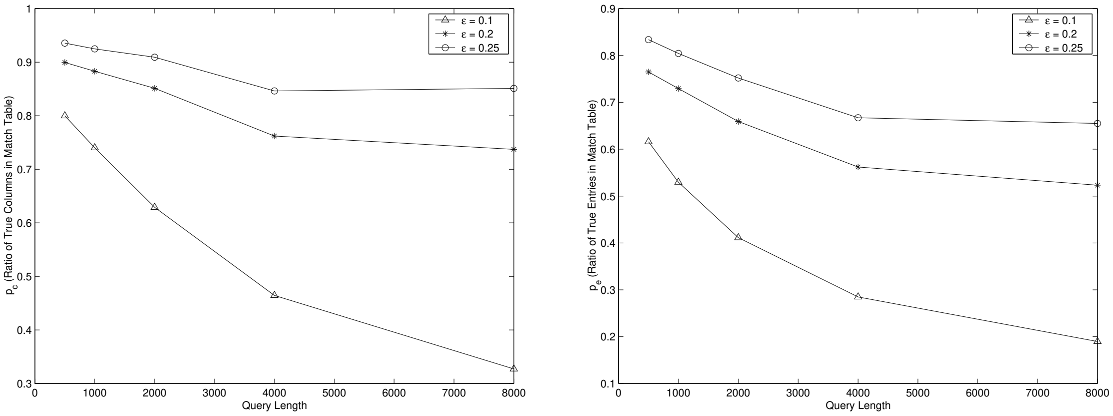

Figure 16. The ratio of *True* columns and entries of the Match table for various query lengths and for different values of ϵ when $\Delta = w =$ maximum possible resolution.

$|q| = 8000$, the memory requirement of MAP is 30 to 40 times less than that of BLAST. This reduction is primarily due to iterating through the database on an MBR by MBR basis.

6.3 Performance comparison

We begin by measuring the ratios p_e and p_c, defined to be ratios of *True* entries and *True* columns in the Match Table. Figure 16 displays these ratios for $\epsilon = \{0.1, 0.2, 0.25\}$ for different query lengths when $\Delta = w$. For $\epsilon = 0.1$, only 33-80% of the Match Table columns contain *True* entries. According to the same figure, only 20-60% of the Match Table entries are marked as *True* for $epsilon = 0.1$.

Figure 17 shows the I/O cost of BLAST and Local MAP for $|q|$ = 8000 and $\Delta = w$ = maximum possible resolution, when memory size is 2%, 10%, 25%, 50%, and 100% of the hash table size. Since the minimum memory requirement of MAP is less than 2% of the hash table size, it reads the required pages into memory once for each using optimal disk scheduling. On the other hand, BLAST incurs numerous of page faults and performs random disk I/Os for small amounts of memory, thus increasing the I/O cost. When the memory is 2% of the hash table, I/O cost of Local MAP is 12 to 32 times less than that of BLAST. When memory is 100% of the hash table (or equivalently infinite amount of memory), I/O cost of Local MAP is still less than that of BLAST. This is because MAP reads only pages corresponding to *True* columns of the Match Table while BLAST reads all of them.

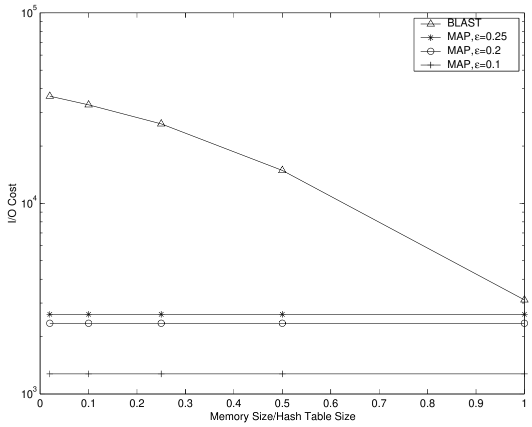

Figure 17. Total I/O cost of BLAST and MAP for different memory sizes when $|q|$ = 8000 and $\Delta = w$ = maximum possible resolution.

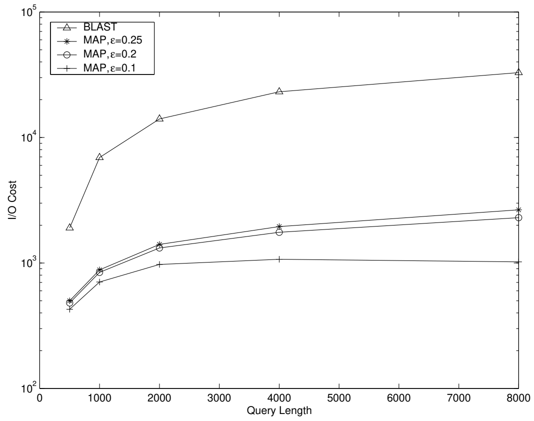

Figure 18. Total I/O cost for BLAST and MAP for different values of $|q|$, when $\Delta = w$ = maximum possible resolution, and the size of memory is 10% of the hash table size.

Figure 18 shows the I/O cost of BLAST and Local MAP for different length queries when $\Delta = w$ = maximum possible resolution, and memory size is 10% of the hash table size. The total I/O cost of MAP increases as ϵ increases since the ratio of *True* is higher for larger values of ϵ. When ϵ = 0.1, I/O cost of MAP is 4 to 32 times less than that of BLAST. When ϵ = 0.25, I/O cost of MAP is 4 to 12 times less than that of BLAST. These results justify the analysis in Section 5.2 that the amount of disk I/O decreases as the ratio of *True* columns decreases.

Figure 19 plots the number of seeds generated for BLAST and MAP for the Match Tables constructed for Figure 16. The number of seeds corresponds to the CPU cost. For $\epsilon = 0.1$, the number of seeds for MAP is one-fifth of that of BLAST, and 45% less than that of BLAST for $\epsilon = 0.25$. These results confirm the analysis in Section 5.3, that the CPU cost decreases as the ratio of *True* entries decreases.

Figure 20 presents the total cost of BLAST and Local MAP expressed in units of sequential page transfers for $|q|$ = 8000 and $\Delta = w$ = maximum possible resolution when memory size is 2%, 10%, 25%, 50%, and 100% of the hash table size. We compute the total cost synthetically as the sum of I/O cost and CPU cost. We converted I/O cost and CPU cost to sequential page transfer cost. The total cost of MAP is 3 to 81 times less than that of BLAST for $\epsilon = 0.1$, and 1.2 to 70 times less than that of BLAST for $\epsilon = 0.25$.

Figure 21 shows the total cost of BLAST and Local MAP expressed in units of page transfers for I/O and CPU costs in Figures 18 and 19. As it can be seen from the figure, the total cost of MAP is 4 to 30 times less than that of BLAST for

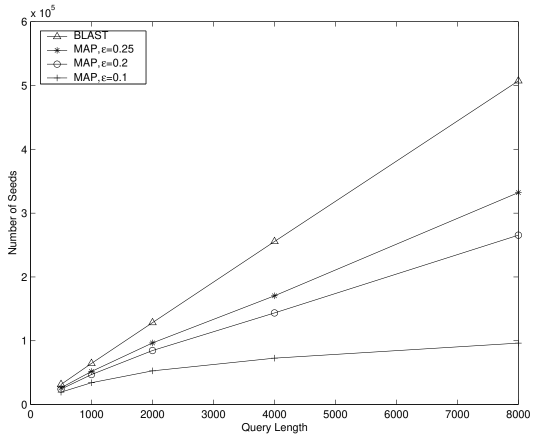

Figure 19. The number of seeds for BLAST and MAP for different values of $|q|$, when $\Delta = w$ = maximum possible resolution.

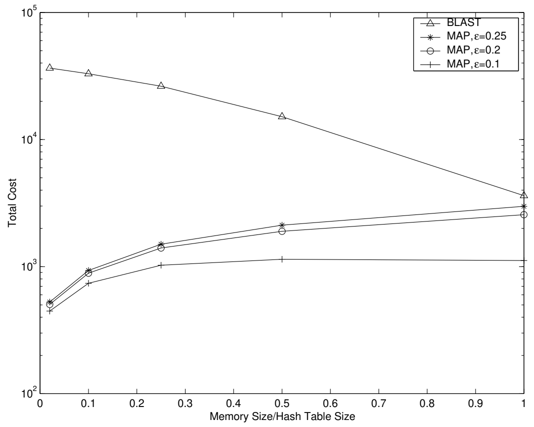

Figure 20. The total cost of BLAST and MAP in terms of page transfers for different memory sizes when $\Delta = w$ = maximum possible resolution.

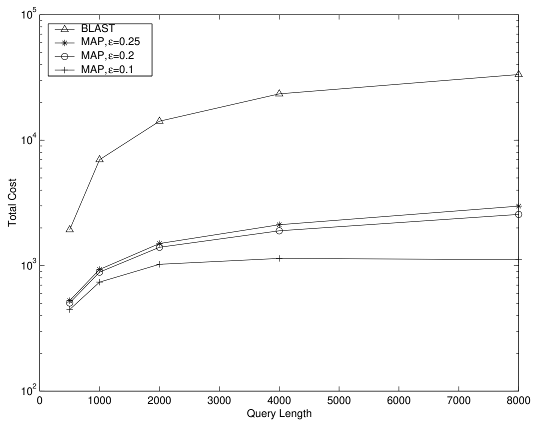

Figure 21. The total cost of BLAST and MAP in terms of page transfers, for different values of $|q|$, when $\Delta = w$ = maximum possible resolution and the size of memory is 10% of the hash table size.

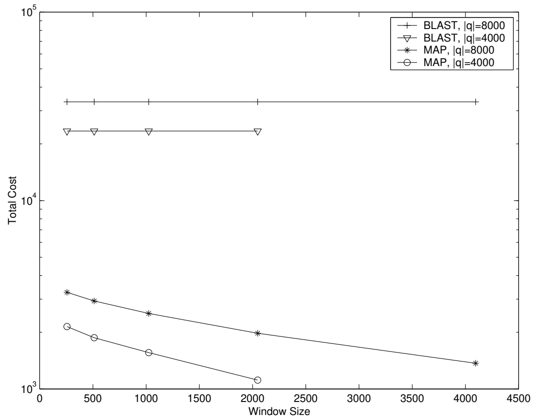

Figure 22. The total cost of BLAST and MAP in terms of page transfers, for different values of w, when $|q| = 4000$ and $|q| = 8000$, and $\Delta = w$, $\epsilon = 0.1$, and the size of memory is 10% of the hash table size.

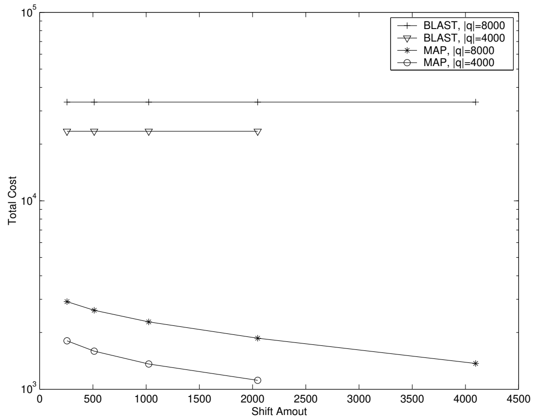

Figure 23. The total cost of BLAST and MAP in terms of page transfers, for different values of Δ, when $|q| = 4000$ and $|q| = 8000$, w = maximum possible resolution, $\epsilon = 0.1$, and the size of memory is 10% of the hash table size.

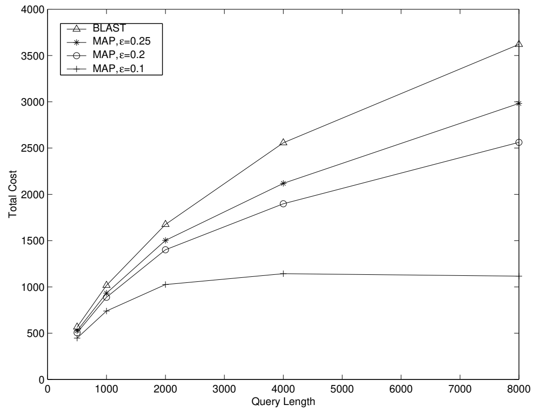

Figure 24. The total cost of BLAST and MAP in terms of page transfers, for different values of $|q|$, when $\Delta = w$ = maximum possible resolution and BLAST is run for each MBR separately.

$\epsilon = 0.1$, and 4 to 11 less than that of BLAST for $\epsilon = 0.25$.

Figure 22 displays the total cost of BLAST and MAP for different window sizes for $|q|$ = 4000 and 8000 when the size of memory is 10% of the hash table size. The total cost of MAP increases as w decreases. This is because of two reasons: First, the relative size of the MBRs increases as w decreases since the data space decreases as w decreases. As a result of this, the likelihood that the frequency vector of the query sequence falls into an MBR increases. Second, since the number of subqueries increases as w decreases, the probability that at least one subquery is similar to a substring in an MBR increases. The experimental results show that the total cost of MAP is 10 to 25 times less than that of BLAST.

Figure 23 plots the total cost of BLAST and MAP for different values of Δ for $|q|$ = 4000 and 8000 when the size of memory is 10% of the hash table size. The total cost of MAP increases as Δ decreases. This is because, as Δ decreases, the number of subqueries increases. As a result of this, the probability that at least one subquery is similar to a substring in an MBR increases. According to the experimental results in figure 23, the total cost of MAP is 11 to 25 times less than that of BLAST.

Our last experiment inspects the performance improvement gained by using a Match Table if BLAST is also run for each MBR separately. In this experiment a separate hash table is constructed for each MBR for BLAST, and BLAST is iteratively run for the MBRs. This guarantees that BLAST reads each database block only once (i.e. it does not perform random disk I/Os.). Figure 24 shows the total cost for different query lengths. As evident from the figure, MAP is always faster than BLAST. The speedup is 3 for $\epsilon = 0.1$, 1.4 for $\epsilon = 0.2$, and 1.17 for $\epsilon = 0.25$. This speedup shows the performance gained by pruning the database using the Match Table.

We can summarize the experimental results as follows.

- The output quality of MAP is very close to that of BLAST.
- MAP is 10 to 30 times faster than BLAST.
- The memory requirements of the MAP is 30 to 40 times less than of that of BLAST.

7 Discussion

In this paper, we considered the problem of speeding up and reducing the memory requirements of substring-substring queries in general and genome queries in particular. We presented an algorithm 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.

With the help of the index structure, the database is organized as a set of MBRs. These MBRs are then iteratively searched one at a time. 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 MRS index structure [26] and the BLAST [1] search tool. The resulting search technique, called MAP, performed much better than BLAST. According to our experimental results, the total cost of MAP is 10 to 30 times less than that of BLAST. The maximum memory requirements of MAP is 30 to 40 times less than that of BLAST. These performance improvements are achieved while keeping quality of the resulting answer set very close to that of BLAST.

Another nice feature of using our technique is that it gives the user the freedom to set quality cutoff, window size, and shift amount parameters to decide on the speed versus quality tradeoff. If the user seeks higher output quality in high scoring regions, this can be done by decreasing the shift amount parameter. If the user seeks higher output quality in all regions, this can be done by decreasing the window size parameter. If the user is only concerned about performance then the shift amount and the window size can be increased and quality cutoff decreased.

We also made significant improvements to the MRS index structure by defining a new notion of distance, and by extending it to work with affine gaps and alphabet specific scores.

In summary, string searching and genome searching are important emerging classes of applications. 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.

References

[1] S. Altschul and B. W. Erickson. Optimal sequence alignment using affine gap costs. *J. Molecular Biology*, 1986.

[2] S. Altschul and W. Gish. Basic local alignment search tool. *J. Molecular Biology*, 1990.

[3] R. A. Baeza-Yates and G. Navarro. A practical index for text retrieval allowing errors. In *CLEI*, volume 1, pages 273–282, November 1997.

[4] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[5] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. In *CPM*, pages 185–192, 1992.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, Atlantic City, NJ, 1990.

[7] D.A. Benson, M.S. Boguski, D.J. Lipman, J. Ostell, and B.F. Ouellette. GenBank. *Nucleic Acids Research*, 26(1):1–7, 1998.

[8] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, B.A. Rapp, and D.L. Wheeler. GenBank. *Nucleic Acids Research*, January 2000.

[9] B. S. Berchtold, C. Bohm, and H.P. Kreigel. The pyramid technique: Towards breaking the curse of dimensionality. In *SIGMOD*, pages 142–153, 1998.

[10] B. S. Berchtold, D. A. Keim, and H. P. Kreigel. The X-tree: An index structure for high dimensional data. In *VLDB*, 1996.

[11] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, and M. Vingron . Rivalsd and. q-gram based database searching using a suffix array (QUASAR),. In *RECOMB*, Lyon, April 1999.

[12] K Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *ICDE*, pages 440–447, February 1999.

[13] A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, and S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[14] P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *SODA*, pages 269–278, Washington, DC, 2001.

[15] E. Giladi, M. G. Walker, J. Z. Wang, and W. Volkmuth. SST: An algorithm for searching sequence databases in time proportional to the logarithm of the database size. In *RECOMB*, Japan, 2000.

[16] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, Roma, Italy, September 2001.

[17] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1 edition, January 1997.

[18] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[19] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, Zrich, September 1995.

[20] J. G. Henikoff and S. Henikoff. BLOCKS database and its applications. *Methods in Enzymology*, 266:88–105, 1996.

[21] S. Henikoff and J. G. Henikoff. Automated assembly of protein blocks for database searching. *Nucl. Acids Res.*, pages 6565–72, 1991.

[22] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. In *PNAS*, pages 10915–10919, 1992.

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

[24] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *VLDB*, pages 139–148, Roma, Italy, September 2001.

[25] H. V. Jagadish, Nick Koudas, and Divesh Srivastava. On effective multi-dimensional indexing for strings. In *SIGMOD*, 2000.

[26] T. Kahveci and A. Singh. An efficient index structure for string databases. In *VLDB*, pages 351–360, Roma, Italy, September 2001.

[27] S. Kurtz and C. Schleiermacher. REPuter - fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5), 1999.

[28] U. Manber and E. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[29] S. Muthukrishnan and S. C. Sahinalp. Approximate nearest neighbors and sequence comparison with block operations. In *STOC*, Portland, Or, 2000.

[30] E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, pages 251–266, 1986.

[31] E. Myers. A sublinear algorithm for approximate keyword matching. *Algorithmica*, pages 345–374, 1994.

[32] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Molecular Biology*, 48:443–53, 1970.

[33] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci*, volume 85, pages 2444–2448, 1988.

[34] P. A. Pevzner and M. S. Waterman. A fast filtration algorithm for the substring matching problem. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *CPM*, pages 197–214, Padova, Italy, 1993. Springer-Verlag, Berlin.

[35] N. Katayama Aand S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *SIGMOD*, pages 369–380, 1997.

[36] B. Seeger, P.-A. Larson, and R. McFayden. Reading a set of disk pages. In *VLDB*, pages 592–603, 1993.

[37] Bernhard Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, 21(5):387–407, 1996.

[38] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, Brighton, England, 1987.

[39] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. of Molecular Biology*, March 1981.

[40] D. J. States and P. Agarwal. Compact encoding strategies for DNA sequence similarity search. In *ISMB*, 1996.

[41] T.A. Tatusova and T. L. Madden. BLAST 2 sequences, a new tool for comparing protein and nucleotide sequences. *FEMS Microbiol. Lett.*, pages 247–250, 1999.

[42] S. Uliel, A. Fliess, A. Amir, and R. Unger. A simple algorithm for detecting circular permutations in proteins. *Bioinformatics*, 15(11):930–936, 1999.

[43] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *ICDE*, pages 516–523, 1996.

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

[45] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *J. of Computational Biology*, 7(1-2):203–214, 2000.