SST: An algorithm for searching sequence databases in time proportional to the logarithm of the database size.

Eldar Giladi [*], Michael G. Walker [†], James Ze Wang [‡] and Wayne Volkmuth [§]

September 30, 1999

Abstract

We have developed an algorithm, called SST (Sequence Search Tree), that searches a database of DNA sequences for near exact matches, in time proportional to the logarithm of the database size n. In SST, we partition each sequence into fragments of fixed length called "windows" using multiple offsets. Each window is mapped into a vector of dimension 4^k which contains the frequency of occurrence of its component k-tuples, with k a parameter typically in the range $4-6$. Then we create a tree-structured index of the windows in vector space, using tree structured vector quantization (TSVQ). We identify the nearest-neighbors of a query sequence by partitioning the query into windows and searching the tree-structured index for nearest neighbor windows in the database. This yields an $O(\log n)$ complexity for the search. SST is most effective for applications in which the target sequences show a high degree of similarity to the query sequence, such as assembling shotgun sequences or matching ESTs to genomic sequence. The algorithm is also an effective filtration method. Specifically, it can be used as a preprocessing step for other search methods to reduce the complexity of searching one large database against another. For the problem of identifying overlapping fragments in the assembly of 120,000 fragments from a 1.5 megabase genomic sequence, SST is 17 to 35 times faster than BLAST when we consider both building and searching the tree. For searching alone (i.e., after building the tree index), SST is 50 to 100 times faster than BLAST.

1 Introduction

In the current efforts to generate and interpret the complete genome sequences of humans and model organisms, large scale searches for near exact matches are frequently performed. Examples include programs that assemble DNA from shotgun sequencing projects which initially search for overlapping fragments, large scale searches of EST databases against genomic databases to determine the location of genes, and cross species genomic comparisons between very closely related genomes. Faster algorithms are needed because of the time and cost of performing these large-scale sequence-similarity searches using even the fastest of the extant algorithms.

[*] Incyte Pharmaceuticals
[†] Incyte Pharmaceuticals and Stanford University
[‡] Stanford University
[§] Incyte Pharmaceuticals

1.1 Previous related research

We now review previous results related to the SST algorithm for sequence alignment, tree-structured indexes, and k-tuple encoding and filtration. In this discussion we shall refer to the length of a query sequence by the letter "m". The size of the database refers to the sum of the lengths of all the sequences in the database, and is represented by the letter "n".

1.1.1 Sequence alignment

Extant widely-used sequence-similarity-finding programs include Needleman -Wunsch [1], Smith-Waterman [2], FASTA [3, 4], and BLAST [5, 6].

The Needleman-Wunsch algorithm performs global sequence alignment using a dynamic programming algorithm. Its computational complexity is $O(m * n)$.

The Smith-Waterman algorithm is a heuristic approximation to the Needleman-Wunsch algorithm which identifies regions of local sequence similarity. Its computational complexity is $O(m * n)$, with a coefficient of complexity smaller than Needleman-Wunsch.

The FASTA algorithm identifies regions of local sequence similarity by first identifying candidate similar sequences based on shared k-tuples and performs local alignment with the Smith-Waterman algorithm. Its computational complexity is $O(m * n)$, with a constant smaller than the Needleman-Wunsch or Smith-Waterman algorithms.

The BLAST algorithm identifies regions of local sequence similarity by first identifying candidate similar sequences that have k-tuples in common with the query sequence, and then extending the regions of similarity. Its computational complexity is $O(n)$.

Myers implemented a sub-linear algorithm that finds long matches with less than a specified fraction of errors [7]. Chang and Lawler also implemented a sub-linear expected-time search algorithm [8]. These algorithms index the occurrences of k-tuples in the database, giving sub-linear complexity. Because each tuple of the query may occur in many sequences in the database these algorithms still require examination of a substantial portion of the sequence database, and the search time still grows linearly with the size of the database. The developers of BLAST evaluated an indexing scheme similar to that described by Myers, but chose not to include it in the final versions of the program [5, 6].

1.1.2 Sequence clustering

Several previous researchers have created clusters of sequences (including tree-structured indexes) or have performed other complete pairwise comparisons of sequence databases [9, 10, 11, 12, 13, 14]. With the exceptions discussed below, these earlier algorithms have used conventional sequence alignment methods such as BLAST or FASTA (rather than the k-tuple method used in SST) to determine pairwise distances between sequences, and thus are limited by the speed of those alignment methods. In most cases, these previous researchers were concerned with creating a classification of proteins, rather than with enabling fast searches.

To perform an exhaustive pairwise search of all sequences in the protein sequence database, Gonnett et al. created a tree-structured index of all the protein sequences and all their subsequences (a Patricia tree), and used Needleman-Wunsch as their measure of similarity between sequences [9]. The size of the resulting tree, and the use of Needleman-Wunsch as the similarity measure, resulted in a computationally intensive algorithm.

Yona and colleagues created hierarchical clusters of the known protein sequences by k-means clustering and metric space embedding [14]. Their intent was to build a global view of protein sequence space, rather than to enable fast searches for similar sequences. In particular, the metric space embedding scheme in their algorithm is computationally intensive, and would be impractical for fast sequence similarity searches.

1.1.3 Tree-structured indexes

Tree-structured indexes have been used previously to provide fast access to databases, and to allow compression of data for rapid transmission. Tree-structured vector quantization (TSVQ) uses a tree-structured index to encode vectors that can be transmitted more quickly than the original data [15], albeit with some loss of information (lossy compression). TSVQ can be viewed as a nearest-neighbor search algorithm, and was the inspiration for the tree-structured index used in SST. Agrawal et al. describe a method to reduce a high-dimensional set of vectors into a lower-dimensional set that retains most of the information, which can then be used to create a tree-structured index into the database [16]. Vector representations and tree-structured indexes have been used widely for database access and data transmission, but to our knowledge these methods have not been used for DNA and protein sequence databases, except for the work of Gonnett and of Yona described above.

1.1.4 K-tuple encodings and filtration

K-tuple encodings have been used frequently in sequence analysis and in conjunction with filtration methods. We have already noted their use in FASTA and in BLAST, and describe here their use in FLASH, RAPID, and QUASAR. The FLASH algorithm [17] uses a form of k-tuple encoding that provides multiple indexes into the database per position in the sequence. The tuples used in FLASH are very descriptive and have a low probability of matching randomly in the database. This yields a high efficiency in screening for candidate database matches.

The RAPID algorithm [18] finds k-tuple matches in each sequence compared to the query sequence (and hence is linear in complexity). Instead of performing a (computationally expensive) alignment as is done in BLAST, RAPID uses a table of word-frequencies computed from the database of sequences and calculates the probability that a set of matching words occur by chance. This approach yields a smaller coefficient of complexity than occurs in BLAST, but also remains linear.

The QUASAR algorithm [19] partitions the database into blocks of equal length, analogous to the "windows" used in SST. It then searches each database block for all occurrences of each k-tuple from the query (q-gram, in their nomenclature), and increments a counter for each block whenever it contains a k-tuple from the query. A lower-bound on the minimum number of k-tuples in a block provides a filter to identify blocks that are similar to the query sequence. The number of blocks which need to be accessed for every q-gram can be linear in the database size.

The use of k-tuple frequency for approximate matching of texts, and in filtration schemes has also been used and analyzed in [20, 21, 22, 23].

SST differs from the previous algorithms in that it combines an efficient embedding of sequence fragments (windows) into metric space, using k-tuple frequency encoding, with a tree-structured index for that space. SST's tree-structured index eliminates large portions of the database from consideration during searches, and requires us to search only a small fraction of the database to find sequences with high similarity to the query sequence. The construction of the index has an average computational complexity which is $O(n \log n)$ while the search of the index has an average complexity $O(m \log n)$.

2 The SST algorithm

We now describe the details of the SST algorithm. Sections 2.1, 2.2 and 2.3 describe the steps in constructing the tree index. These steps need to be performed only once for each database. Section 2.4 describes how the database index is searched.

2.1 Data-base partitioning with sliding windows

In the first step each sequence in the database is partitioned into overlapping windows, which consist of subsequences of nucleotides of fixed length W. The measure of overlap between windows is determined by a parameter Δ which is typically in the range $5 \leq \Delta \leq W/2$. The windows consist of the nucleotides beginning at position $j * \Delta$, and ending at position $j * \Delta + W$, with $j = 0, 1, 2 \ldots, W/\Delta - 1$ (Figure 1). Typical values of W are in the range $25 - 1000$.

Figure 1: A database sequence is partitioned into overlapping windows of length $W = 6$. The overlap parameter $\Delta = 2$.

Each query sequence is partitioned into *non-overlapping* windows. The search for a query sequence consists of finding database windows that are similar to the query windows, as will be described in the next subsections.

2.2 Mapping windows into vector space

In order to build the tree index used by SST, it is necessary to map each window into a finite dimensional vector space. We begin by choosing a tuple size k, and by associating an integer I with each of the 4^k k-tuples of nucleotides $a_1 a_2 \cdots a_k$, $a_i \in \{\text{A,C,G,T}\}$. This is done in a standard fashion using the formula

$$I(a_1 a_2 \cdots a_k) \ = \ \sum_{l=1}^{k} 4^l M(a_l),$$

$$M(a_l) = \begin{cases} 0 & \text{if } a_l = \text{A}, \\ 1 & \text{if } a_l = \text{C}, \\ 2 & \text{if } a_l = \text{G}, \\ 3 & \text{if } a_l = \text{T}. \end{cases}$$

Tuples containing the undetermined symbol "N" are ignored. We now map each database window to the vector in R^{4^k} who's I'th entry is the number of occurrences of tuple I in that window. For example, with $k = 2$ the window $\{$AACCGG$\}$ in Figure 1 is mapped to the vector

$$(1100011000100000)^T.$$

The tuple size k ranges between 2 and 10 but is typically 4 or 5, chosen for reasons described in Section 4.

The L_1 distance (Manhattan distance) between the image of two windows in vector space is the number of k-tuples which occur in one of the windows and not in the other. Since each window has $W - k + 1$ component k-tuples, their distance can be used to determine the number of k-tuples shared by the two windows. This in turn provides a lower bound for the edit distance between the windows [23], [22]. The correspondence between the L_1 distance in vector space and the edit distance in sequence space is utilized in SST to find near exact matches to query windows, by seeking nearest neighbors in metric space.

We note that the mapping procedure described in this section is easily modified to deal with other alphabets such as the 20 letters used for protein sequences or a reduced alphabet.

2.3 Tree structured index for database windows

The mapping described in the previous section transformed the problem of finding near exact matches to query windows in the database to that of finding nearest neighbors in vector space. This search can be done very efficiently by constructing a tree-structured index in vector space.

To create the tree-structured index, we recursively search the data for clusters that provide binary (or higher-order) partitions. A variety of methods are available for finding such clusters and building the tree. One suitable method is Tree-Structured Vector Quantization [15] using k-means clustering which we have used and describe here.

1. Select two centroids x_A, x_B and their corresponding partitions of the data into disjoint sets A and B using the following iterative procedure
   - Choose two initial values for x_A and x_B.
   - For each vector y in the database compute the L_1 distances

     $$d_A(y) = \|x_A - y\|_1, \quad d_B(y) = \|x_B - y\|_1.$$

     Assign y to the set A if $d_A(y) < d_B(y)$, and to the set B otherwise.
   - Compute

     $$D_A = \sum_{y \in A} d_A(y), \quad D_B = \sum_{y \in B} d_B(y).$$

   - Compute the new centroids

     $$x_A = \frac{\sum_{y \in A} y}{|A|}, \quad x_B = \frac{\sum_{y \in B} y}{|B|}.$$

- Repeat until the change in D_A and D_B is less than a small threshold, or no vectors change partition.

2. Recursively partition the sets A and B generated above using the same algorithm.

3. The recursion terminates when the number of vectors in a set is smaller then a specified tolerance or when the algorithm fails to fragment a cluster into two substantial new clusters.

The leaves of the tree partition the k-dimensional space, and each leaf contains the set of vectors that are nearest-neighbors to the centroid for that node. We note that the depth of the tree is proportional to the logarithm of the number of windows and the number of windows is proportional to the size of the database. Hence the average complexity of the tree construction is $O(n \log n)$.

2.4 The search procedure.

For each query, the SST algorithm finds similar database sequences by searching for database windows that are similar to at least one of the query windows. This search is done efficiently in vector space using the tree structured index as follows.

Begin at the root of the tree. At each non-terminal node there are two branches (for the case of a binary tree), which are represented by their respective centroid. Select the branch whose centroid is the lesser distance from the query vector. Proceed recursively until reaching a terminal node. The vectors in the terminal node represent the database windows which are the nearest neighbors to the query window.

The SST algorithm finds most of the nearest neighbors but is not guaranteed to find all the nearest neighbors of a query sequence. In particular, sequences that lie near the boundary of a partition may be closer to another sequence immediately across the boundary line than they are to any other sequence within the partition; we indicate the false negative rate for one experiment below.

Additional processing such as alignment of the query to the database sequence with one of the standard alignment tools is possible.

3 Computational implementation

We now describe computer implementation strategies for the SST algorithm which significantly improve the speed and the scalability of the algorithm. These strategies include computation on sub-trees, sampling in the construction of the k-means tree, the use of sparse vector representation, finite precision arithmetic and a compressed format for the database windows.

As the number of sequences increases, it becomes impossible to store the whole tree index in RAM. To minimize disk access during the computation we perform the tree construction and search on sub-trees. For example, when searching one database against another, we first search all query windows against the top part of the tree, say the first 9 levels. This generates $2^9 = 512$ groups of windows. Now, each group is searched against its respective subtree. Disk access occurs only once for each subtree because it is small enough to fit into RAM. The subtree approach can be generalized to an arbitrary number of levels. Moreover, it provides a foundation for parallelizing the algorithm because adjacent subtrees can be processed independently on different processors.

The speed of the tree construction can be substantially enhanced by computing the centroids based on a sample of the windows in each cluster, rather than using all of them. We find that a sample of 1000 sequences to estimate two centroids provides a substantial improvement in the speed at a relatively low cost in the error rate; where higher accuracy is required, larger samples are useful.

The computation of the centroids requires the use of floating point arithmetic. However, floating point operations are more expensive than integer arithmetic. Hence, in our implementation we use 16 bit finite precision arithmetic.

All frequency vectors are very sparse. They have at most $W - k + 1$ non-zero entries while their dimension is 4^k. Typical values are $W = 50$, $k = 5$, $4^5 = 1024 \gg 46$. The use of sparse vector representation allows us to store only the non-zero entries, by storing pairs (index, value), instead of storing the whole vector. This representation saves a substantial amount of space in both RAM and disk. In addition, substantial computations are saved by using sparse vector arithmetic, for example in the computation of the distance of each vector from a centroid.

Database windows overlap and therefore have large portions in common. We use a compressed format for the windows which takes advantage of this overlap. The storage required for all windows of a database in this format is independent of both the window size W and the offset size Δ. The storage required by this format is about 2 bytes per nucleotide.

4 Application of SST to shotgun sequence assembly

We now illustrate the performance of SST by applying it to the task of detecting overlapping fragments in shotgun sequence assembly. We also compare the speed of SST to BLAST in order to highlight the computational complexity of SST.

In our simulations, we fragmented a 1.5 megabase piece of genomic DNA (sequenced by the Human Genome Center Institute of Medical Science University of Tokyo, contig CN00029) several times using a Poisson process with $\lambda = 300$ nucleotides. From the pool of fragments we generated three sets of 30675, 61350 and 122700 sequences, respectively, thus simulating 6-fold, 12-fold, and 24-fold coverage of the 1.5 megabase genomic DNA. Fragments smaller than 50 base pairs were rejected as is the practice in sequence assembly and fragments larger than 350 base pairs were also rejected to simulate a common length limit in sequencing. We then used SST and BLAST2 (with no gaping option, all other parameters taken as default values) to determine which of the sequences overlap.

In the computation with SST, we used a window size $W = 50$ and an offset step $\Delta = 5$. We repeated the computation for several values of the tuple size k, specifically $k = 3, 4, 5, 6$. Table 1 presents the results of SST on each of these sets. We report the number of sequences, tuple size k, the time required to construct the tree ("Construction"), the time required to search the tree using every sequence as a query sequence ("Query"), and the sum of the construction time and the query time ("Total"). We also report the true-positive rate, the false-negative rate, the true-negative rate, and the false-positive rate (determined by comparing putative hits identified by BLAST or SST to the known fragments of the genomic sequence).

A k-tuple size of 3 yields the fastest time for construction and query, but a k-tuple size of 6 yields the smallest error rate. We see that, for SST, the times for a complete pairwise search of the database scale linearly with the number of sequences. We also note that the true negative rate is, to five decimal places, 1 in all cases. This highlights the effectiveness of SST as a filtration scheme.

# Seqs	K	Total	Construction	Query	TPR	FNR	TNR	FPR
30675	3	00:10:12	00:06:56	00:03:16	0.893	0.107	0.99998	2.092e-05
61350	3	00:22:33	00:15:36	00:06:57	0.886	0.114	0.99998	1.842e-05
122700	3	00:46:56	00:31:52	00:15:04	0.881	0.119	0.99998	1.529e-05
30675	4	00:11:37	00:07:24	00:04:13	0.932	0.068	0.99999	1.069e-05
61350	4	00:26:11	00:17:37	00:08:34	0.926	0.074	0.99999	9.805e-06
122700	4	00:50:50	00:32:50	00:18:00	0.924	0.076	0.99999	8.404e-06
30675	5	00:13:52	00:08:24	00:05:28	0.956	0.044	0.99999	6.642e-06
61350	5	00:36:49	00:19:00	00:17:49	0.953	0.047	0.99999	6.222e-06
122700	5	01:08:22	00:35:21	00:33:01	0.951	0.049	0.99999	5.950e-06
30675	6	00:23:04	00:13:04	00:10:00	0.963	0.037	0.99999	5.284e-06
61350	6	00:45:53	00:26:37	00:19:16	0.960	0.040	0.99999	5.018e-06
122700	6	01:34:50	00:52:21	00:42:29	0.957	0.043	1.0000	4.793e-06

Table 1: SST results with 30675, 61350 and 122700 fragments, simulating a 6-fold, 12-fold and 24-fold coverage, respectively. The time for tree construction, tree search and total time are displayed for $k = 3 - 6$. The true positive rate (TPR), false negative rate (FNR), true negative rate (TNR), and false positive rate (FPR) are also displayed.

Table 2 shows the time to search the data with BLAST2 versus SST. We choose a k-tuple size of 5 as a reasonable compromise between speed and error rate. For overlaps of size 50, as used in this experiment, BLAST2 has 100 percent true positive and true negative rates. We see that, for BLAST2, the times for a complete pairwise search of the database scale quadratically with the number of sequences.

# of fragments	Blast search time	BLAST time per query	SST time per query
30675	02 : 03 : 09	0.2409	0.0164
61350	07 : 27 : 47	0.4379	0.0186
122700	28 : 10 : 25	0.8266	0.0173

Table 2: The time to determine overlapping sequences using BLAST and SST. The third and fourth columns is the time per query for BLAST and SST respectively.

It is useful to consider the average search time per sequence for SST versus BLAST2. Columns three and four of Table 2 shows these data for BLAST2, and SST respectively as a function of the number of sequences in the database. We see a linear increase in search-time per sequence with BLAST2 as the database grows, while the search-time per sequence for SST is nearly constant. Thus, search time per sequence is nearly independent of the database size with the SST algorithm. Figure 2 depicts this information graphically.

5 Discussion

SST is most effective for applications in which the target sequences show a high degree of similarity to the query sequence, such as assembling shotgun sequences or matching ESTs to genomic sequence. For some applications, such as assembly of shotgun sequences, it is sufficient to identify the set of nearest-neighbor sequences. For other applications, it is desirable or necessary to align the nearest-neighbor sequences to the query sequence using an algorithm such as Needleman-Wunsch or Smith-Waterman. Since SST filters most true negatives only a small number of sequences are returned for each query, and such alignments can be done relatively quickly.

In the example presented here, for 120,000 sequences, SST is 25 to 50 times faster than BLAST 2. Because SST scales logarithmically with the number of sequences, while BLAST2

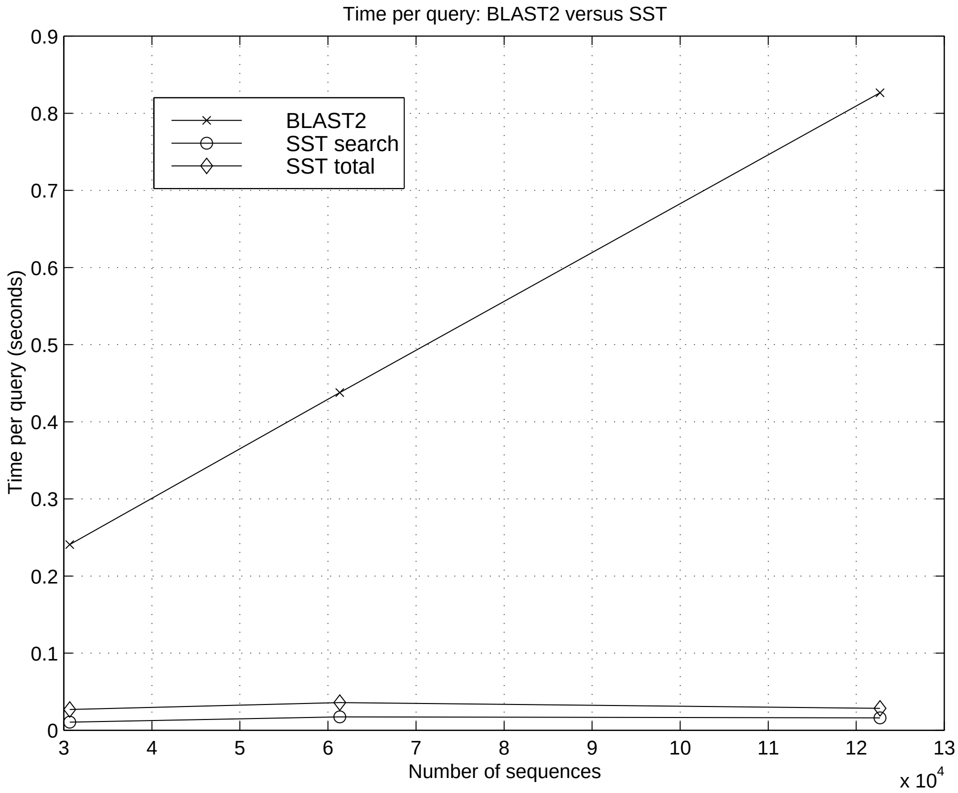

Figure 2: The ordinate is the time per query and the abscissa is the number of sequences. The two timings for SST are for query only and for query and construction.

scales linearly, we estimate that for a database of one million sequences SST will be 250 to 500 times faster than BLAST2, with similar gains for yet larger sequence collections.

6 Acknowledgements

We wish to thank Junming Yang, Tod Klingler and Richard Goold for stimulating discussions on this work.

References

[1] Needleman S. B. and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–53, 1970.

[2] Smith T. F. and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–7, 1981.

[3] Pearson W. R. and D. J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci USA*, 85(8):2444–8, 1988.

[4] Pearson W. R. Effective protein sequence comparison. *Methods Enzymol*, 266:227–58, 1996.

[5] S. F. Altschul, W.Gish, et al. Basic local alignment search tool. *J Mol Biol*, 215(3):403–10, 1990.

[6] Altschul S. F., T. L. Madden, et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–402, 1997.

[7] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374, 1994.

[8] Chang W. and E. Lawler. Approximate string matching in sublinear expected time. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri*. IEEE Computer Society Press, 1990.

[9] Gonnett G. H., M. A. Cohen, et al. Exhaustive matching of the entire protein sequence database. *Science*, 256(5 June):1443–1445, 1992.

[10] Harris N.L., L. Hunter, et al. Mega-classification: Discovering motifs in massive datastreams. In *Proceedings of the 10th National Conference on Artificial Intelligence*. AAAI Press, 1992.

[11] Watanabe H. and J. Otsuka. A comprehensive representation of extensive similarity linkage between large numbers of proteins. *Comput Appl Biosci*, 11(2):159–66, 1995.

[12] Barker W. C., F. Pfeiffer, et al. Superfamily classification in PIR-international protein sequence database. *Methods Enzymol*, 266:59–71, 1996.

[13] Tatusov R. L., E. V. Koonin, et al. A genomic perspective on protein families. *Science*, 278(5338):631–7, 1997.

[14] Yona G. *Methods for global organization of all known protein sequences*. PhD thesis, Computer Science, Hebrew University, May 1999.

[15] Gersho A. and R. Gray. *Vector quantization and signal compression*. Kluwer, Boston, 1992.

[16] Agrawal R., W. R. Equitz, et al. Method for high-dimensionality indexing in a multi-media database. US Patent, Appl. No. 607922, 1997.

[17] Califano A. and I. Rigoutsos. Flash: Fast look-up algorithm for string homology. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1995.

[18] Miller C., J. Gurd, et al. RAPID: an extremely fast dna database search tool. In *Genes, proteins, and computers V*. York University, England, 1998.

[19] Burkhardt S., A. Crauser, et al. Q-gram based database searching using a suffix array (QUASAR). In *Third International Conference on Computational Molecular Biology (Recomb 99), Lyon, France*. ACM Press, 1999.

[20] P. A. Pevzner and M.S. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13:135–154, 1995.

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

[22] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[23] P. Jokinen and E. Ukkonen. Two algorithms for approximate string-matching in static texts. In *Proc. of the 16'th Symposium on Mathematical Foundations of Computer Science*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248, 1991.