BIOINFORMATICS

Vol. 18 no. 6 2002
Pages 873–879

SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size

Eldar Giladi [1,], Michael G. Walker [1], James Z. Wang [2] and Wayne Volkmuth [1]*

[1]Incyte Pharmaceuticals, 3174 Porter Drive, Palo Alto, CA 94304, USA and [2]Department of Computer Science, Pennsylvania State University, University Park, PA 16802, USA

Received on January 11, 2001; revised on January 7, 2002; accepted on January 29, 2002

ABSTRACT

Motivation: Searches for near exact sequence matches are performed frequently in large-scale sequencing projects and in comparative genomics. The time and cost of performing these large-scale sequence-similarity searches is prohibitive using even the fastest of the extant algorithms. Faster algorithms are desired.

Results: 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, with 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. When the tree is balanced this yields an $O(\log n)$ complexity for the search. This complexity was observed in our computations. 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 15 times faster than BLAST when we consider both building and searching the tree. For searching alone (i.e. after building the tree index), SST 27 times faster than BLAST.

Availability: Request from the authors.

Contact: egiladi@incyte.com; mwalker@incyte.com

*To whom correspondence should be addressed.

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 the time and cost of performing these large-scale sequence-similarity searches using even the fastest of the extant algorithms is prohibitive.

1.1 Previous related research

We now review previous results related to the Sequence Search Tree (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 (Needleman and Wunsch, 1970), Smith–Waterman (Smith and Waterman, 1981), FASTA (Pearson and Lipman, 1988; Pearson, 1996) and BLAST (Altschul *et al.*, 1990, 1997).

The Needleman–Wunsch and Smith–Waterman algorithms perform global and local sequence alignment using a dynamic programming algorithm. Their computational complexity is $O(m * n)$.

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 (Myers, 1994). Chang and Lawler also implemented a sub-linear expected-time search algorithm (Chang and Lawler, 1990). 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 (Altschul *et al.*, 1990, 1997).

1.1.2 Sequence clustering. Several previous researchers have created clusters of similar sequences (including tree-structured indexes) or have performed other complete pairwise comparisons of sequence databases (Gonnett *et al.*, 1992; Harris *et al.*, 1992; Watanabe and Otsuka, 1995; Barker *et al.*, 1996; Tatusov *et al.*, 1997; Yona, 1999). 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 (Gonnett *et al.*, 1992). 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 (Yona, 1999). 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 (Gersho and Gray, 1992), 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 (Agrawal *et al.*, 1997). 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 (Califano and Rigoutsos, 1995) 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 (Miller *et al.*, 1998) 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 (Burkhardt *et al.*, 1999) 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

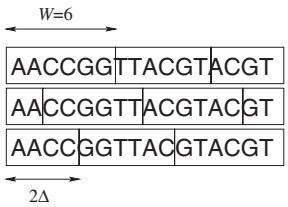

Fig. 1. A database sequence is partitioned into overlapping windows of length $W = 6$. The overlap parameter $\Delta = 2$.

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 analysed (Pevzner and Waterman, 1995; Wu and Manber, 1992; Ukkonen, 1992; Jokinen and Ukkonen, 1991). 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 2.1–2.3 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 Database 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.

Each query sequence is partitioned into *non-overlapping* windows or into windows which overlap by half their length. 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. Specifically, for each sequence, we create a vector consisting of counts of the number of occurrences of each k-tuple. Thus, for k of 2, we create a vector of length 16 that consists of counts of the number of occurrences of the 16 k-tuples AA, AC, AG, AT, CA,..., TG, TT. We begin by choosing a tuple size k, and by associating an integer I with each of the 4^k k-tuples of nucleotides $a_1 a_2 \cdots a_k$, $a_i \in$ {A,C,G,T}. This is done in a standard fashion using the formula

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

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

Tuples containing the undetermined symbol 'N' are ignored. We now map each database window to the vector in R^{4^k} who's Ith 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 (Jokinen and Ukkonen, 1991; Ukkonen, 1992). 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 TSVQ (Gersho and Gray, 1992) 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 from the vector to each of the centroids:

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

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

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

- Compute values for the terminating criteria

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

  where $|A|$ is the size of set A.
- 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 when the tree is balanced 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 dividing the query sequence into windows and 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, fixed 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 subtrees. For example, when searching one database against another, we first search all query windows against the top part of the tree, say the first nine 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 windows 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 a 16 bit fixed 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 $W = 50$, $k = 5$, $4^5 = 1024 \gg W - k + 1 = 46$. 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 parameter Δ. The storage required by this format is about 2 bytes per nucleotide.

When using SST we build one tree for the sequences in the database and one tree for their complement. We then search both trees. These computations are independent and can be performed concurrently.

4 COMPUTATIONAL RESULTS

We illustrate the performance of SST by applying it to detecting overlapping fragments in shotgun sequence assembly. We fragment a 1.5 megabase sequence of genomic DNA several times using a Poisson process with $\lambda = 300$ nucleotides. From the pool of fragments we generate three sets of fragments of: 30 675, 61 350 and 122 700 sequences thus simulating a 6-fold, 12-fold and 24-fold coverage of the genomic stretch. We apply SST to search each database of fragments against itself for overlapping sequences, with overlap size ≥ 50.

In the first computation we randomly introduce errors into each fragment at a rate of 5% with a maximum of 2.5% insertions/deletions. In this computation the window size W and the tolerance T (the distance threshold for deciding if a query sequence matches a database sequence) are varied in the range $30 \leq W \leq 50$ and $0 \leq T \leq 30$. The number of fragments is 61 350 and the tuple size used in the encoding is $k = 5$. Query windows do not overlap.

Figure 2 indicates the true positive rate (TPR) and the $\log_{10}$ of the false positive rate (FPR) as a function of W and T. The TPR increases as W decreases and as T increases, and so does the FPR. With optimal parameters $W = 30$, $T = 15$, the TPR and FPR are .95 and

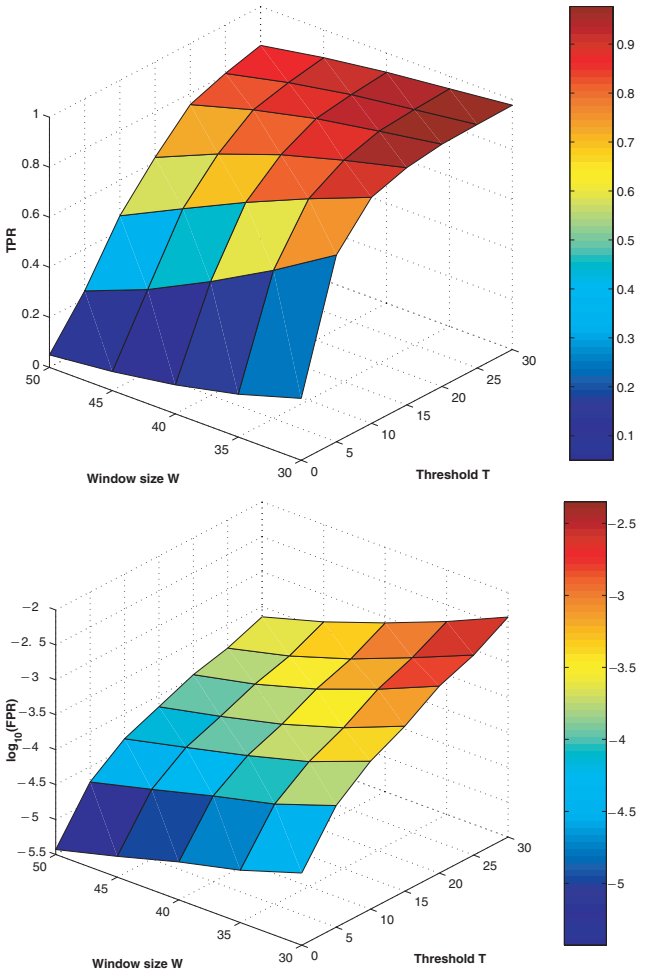

Fig. 2. Top: the TPR as a function of the window size W and the the distance threshold T. Bottom: the **log** base 10 of the FPR as a function of the window size W and the distance threshold T on the right. Data has an error rate of 5%.

0.0007, respectively. In the remainder of this section all computations use a window size of 30 and a threshold distance of $T = 15$.

We now study the effect of varying the tuple size k on the accuracy and the speed of SST. The number of fragments is 61 350 and the error rate is 5% with 2.5% insertions/deletions. The tuple size k varies in the range $3 \leq k \leq 6$. Tables 1 and 2 indicate the results of this test when there is no overlap between query windows and when the overlap is half the window size, respectively. The computation time increases with the tuple size k. The accuracy of the results also improve as k increases until $K = 5$ and then decreases. The results are more accurate when the query windows overlap but the computation is much longer in this case. In the remainder of this section all computations are done with a tuple size of $k = 5$.

Table 1. The total time (to build the tree and search), the time to search the tree, the true positive rate (TPR) and the false positive rate (FPR) as a function of the tuple size K used. There is no overlap between the query windows

K	Total time	Search time	TPR	FPR
3	01:00:36	00:16:15	0.823	1.151e−03
4	01:28:24	00:21:29	0.904	7.746e−04
5	01:28:21	00:33:50	0.941	6.849e−04
6	02:34:55	01:31:23	0.923	5.869e−04

Table 2. The total time (to build the tree and search it), the time to search the tree, the true positive rate (TPR) and the false positive rate (FPR) as a function of the tuple size K used. Query windows overlap by half their length

K	Total time	Search time	TPR	FPR
3	01:08:44	00:24:55	0.928	2.024e−03
4	01:39:45	00:32:44	0.969	1.284e−03
5	01:49:28	00:55:03	0.985	1.110e−03
6	03:47:16	02:43:37	0.977	9.479e−04

We now study the degradation of the performance of the algorithm as the error rate (nucleotide mismatches, insertions, and deletions) increases. Tables 3 indicate the TPR and FPR as a function of the error rate P. The degradation of the performance is apparent. The accuracy improves when query windows overlap.

We expect the TPR and FPR for BLAST to be as good as or better than those for SST. For an application such as sequence assembly, imperfect sensitivity (failure to detect true hits) is similar in effect to sequencing with less coverage; the missing sequences do not contribute to the initial assembly. SST is best used as a filtration method, whereby the small number of sequences returned by SST are confirmed by a slower method such as BLAST. This post-processing provides a method to remove or limit false positives with little computational expense.

We now compare the scaling of the computation time *per query* between SST and BLAST as we vary the database size. Tables 4 and 5 show the linear scaling of the time per query with BLAST versus the nearly constant time with SST. The speed up compared to BLAST is also indicated. We see that for search alone SST is 27 times faster than BLAST while for building the tree index and searching it, SST is about 15 times faster then BLAST for the database of 120 000 sequences when query windows do not overlap. When query windows overlap SST is about 9.3 times faster than BLAST for building the index and searching it and 13.2 times faster than BLAST for searching alone. A higher speed up is expected for larger databases.

Table 3. The true positive rate (TPR) and the false positive rate (FPR) as a function of the error rate P. Top, query windows do not overlap; bottom, query windows overlap. The rate of insertions/deletions is 2.5%

P	TPR	FPR
5	0.941	6.849e−04
10	0.718	3.277e−04
15	0.436	1.565e−04
20	0.202	6.547e−05

P	TPR	FPR
5	0.985	1.110e−03
10	0.857	5.540e−04
15	0.610	2.746e−04
20	0.326	1.197e−04

Table 4. The total time per query (build index and search), the speed up compared to BLAST, the time per query for search alone, the speed up compared to BLAST for search, the BLAST time per query. Query windows do not overlap

Coverage	SST total time	Total time speed up	SST query time	Search time speed up	BLAST time
6	0.0503	4.7	0.0246	9.7	0.2409
12	0.0538	8.1	0.0276	15.8310	0.4379
24	0.0561	14.7	0.0301	27.4	0.8266

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. The accuracy is greatly improved when query windows overlap but this comes at the cost of a substantial slowdown. 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 15 to 27 times faster than BLAST2, when query windows do not overlap. As seen in our computations SST scales logarithmically with the number of sequences, while BLAST2 scales linearly. Hence we estimate that for a database of one million sequences SST will be 170 to 270 times faster than BLAST2, with similar gains for yet larger sequence collections.

Table 5. The total time per query (build index and search), the speed up compared to BLAST, the time per query for search alone, the speed up compared to BLAST for search, the BLAST time per query. Query windows overlap by half their length

Coverage	SST total time	Total time speed up	SST query time	Search time speed up	BLAST time
24	0.0891	9.27	0.0626	13.2	0.8266
12	0.0757	5.78	0.0494	8.86	0.4379
6	0.0704	3.24	0.0444	5.4	0.2409

ACKNOWLEDGEMENTS

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

REFERENCES

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

Altschul,S.F., Madden,T.L. *et al.* (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.

Altschul,S.F., Gish,W. *et al.* (1990) Basic local alignment search tool. *J. Mol. Biol.*, **215**, 403–410.

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

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

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

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

Gersho,A. and Gray,R. (1992) *Vector Quantization and Signal Compression*. Kluwer, Boston.

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

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

Jokinen,P. and Ukkonen,E. (1991) Two algorithms for approximate string-matching in static texts. In *Proceedings of the 16th Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 520, pp. 240–248.

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

Myers,E. (1994) A sublinear algorithm for approximate keyword searching. *Algorithmica*, **12**, 345–374.

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

Pearson,W.R. (1996) Effective protein sequence comparison. *Methods Enzymol.*, **266**, 227–258.

Pearson,W.R. and Lipman,D.J. (1988) Improved tools for biological sequence comparison. *Proc. Natl Acad. Sci. USA*, **85**, 2444–2448.

Pevzner,P.A. and Waterman,M.S. (1995) Multiple filtration and approximate pattern matching. *Algorithmica*, **13**, 135–154.

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

Tatusov,R.L., Koonin,E.V. *et al.* (1997) A genomic perspective on protein families. *Science*, **278**, 631–637.

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

Watanabe,H. and Otsuka,J. (1995) A comprehensive representation of extensive similarity linkage between large numbers of proteins. *Comput. Appl. Biosci.*, **11**, 159–166.

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

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