J Supercomput (2007) 40: 185–212
DOI 10.1007/s11227-006-0041-0

Survey on index based homology search algorithms

Xianyang Jiang · Peiheng Zhang · Xinchun Liu · Stephen S.-T. Yau

Published online: 23 March 2007

Abstract Up to now, there are many homology search algorithms that have been investigated and studied. However, a good classification method and a comprehensive comparison for these algorithms are absent. This is especially true for index based homology search algorithms. The paper briefly introduces main index construction methods. According to index construction methods, index based homology search algorithms are classified into three categories, i.e., length based index ones, transformation based index ones, and their combination. Based on the classification, the characteristics of the currently popular index based homology search algorithms are compared and analyzed. At the same time, several promising and new index techniques are also discussed. As a whole, the paper provides a survey on index based homology search algorithms.

Keywords Algorithm · Bioinformatics · Genomic indexing · Homology search · Sequence alignment

X. Jiang (✉)
IRISA-INRIA, Campus de Beaulieu, 35042 Rennes cedex, France
e-mail: xyjiang@ncic.ac.cn

X. Jiang · P. Zhang · X. Liu
Institute of Computing Technology, CAS, 100080, Beijing, People's Republic of China

S.S.-T. Yau (✉)
MSCS, University of Illinois at Chicago, Chicago, IL 60607-7045, USA
e-mail: yau@uic.edu

S.S.-T. Yau
Institute of Mathematics, East China Normal University, Shanghai, People's Republic of China

1 Introduction

Molecular biologists frequently query genomic databases for sequence homology. One important goal of homology search, sequence matching, or sequence anchoring is to determine whether there are any sequences known in the database similar to a query sequence. If two sequences are very similar by a defined standard, for example, a distance, it is likely that:

- The sequences have a related structure or function. In most cases, there is some available information on the structure and function of the sequences in the database. A scientist with a sequence similar to a known sequence may be able to gain some information about the form and function of the new sequence by studying similar known sequences.
- The sequences may have a common ancestor sequence. If two sequences are reasonably similar, it is likely that both sequences evolved from a common ancestor and an evolutionary relationship may exist between the source of each sequence.
- If the query sequence is a partial sequence, it may be possible to gain information about the sequence's position and role in the sequence which it comes from.

However, homology search or similarity computation has a time complexity of either linear or quadratic to the length of the sequences (or database) involved. Such time complexity is a serious problem for the following states:

- Genomic databases are increasing in size in terms of both number of sequences and length of the sequence, and the size is doubling every 15 or 16 months.
- The number of queries directed at these databases are over 40,000 queries every day, and at the same time, the user numbers and query rates are growing very quickly.
- There is an increasing demand to mine a sequence database for useful information. This typically requires to do all pair wise similarity computation for all the sequences; such job will need a large computing resource and a lot of time.

These factors together increase the need for high computation ability, and if the former successive exhaustive search techniques are not practical or economical now, then it is necessary to invent novel and efficient methods for searching genomic databases.

One developed and promising direction in literature is to generate an abstraction or index for the database sequences or inquiry sequences at first, and then, based on the index, to refine the computation to get a candidate answer. Most indices represent a sequence by its "subsequences" (*motifs*). The "subsequences" can be either real subsequences, or subsequences in a translated meaning (e.g. transformed index).

In most cases, an index is only a fraction of the database, thus the query evaluation costs, i.e., computing time and memory requirements, can decrease. When the total index is not a fraction of the database, the access can still be limited to a fraction of the index which is produced based on a filtering or filtration function.

As to the index based methods for genomic homology search, Navarro [23] et al. in 2001, along two dimensions: data structure and search method, presented a simple classification for the approaches up to that date. Based on the classification, the authors pointed out that the most promising alternatives were those looking for an

optimum balance point between exhaustively searching for neighborhoods of pattern pieces and the strictness of the filtration produced by splitting the pattern into pieces.

Because there are many more new techniques proposed for genomic search since then, the classification should be updated now. On the other hand, until now, no good and total classification for index based homology search algorithms is available in literature. In this paper, we introduce several index models, classify the algorithms, and compare characteristics of the algorithms based on the proposed index models. Based on the classification and comparison, we provide the promising research directions.

There are at least two goals for this survey. First, most of the index based algorithms are motivated by the issues of mathematical description of the sequence, thus the classification of them is possible and we can even invent new techniques along this way. Second, a strong and consistent basis is necessary for index based homology search algorithms. This survey will provide a theoretical base at this point, and although this paper does not accomplish this task, the intention is at least to raise the issue.

The paper is organized as following. In Sect. 2, we introduce the motivation to adopt index for homology search, the characteristics of genomic index, the index models and their construction methods. Then, in Sect. 3 and Sect. 4, the algorithms based on different index models are briefly described respectively. The algorithms using combined techniques are introduced in Sect. 5. Finally in Sect. 6, the conclusion and some discussions are provided.

2 Genomic data structure and index necessity

2.1 Genomic data structure

Most text databases are with hierarchical structure. However, for the genomic data, it is a little different. The most popular genomic databases are GenBank (Gene Bank), DDBJ (DNA Data Bank of Japan), EMBL (European Molecular Biology Laboratory). The bases of these databases are linear DNA sequences and primary protein sequences. DNA sequence is composed of four kinds of nucleic acids, and it is the basic genetic information carrier. Primary protein sequence is composed of 20 kinds of amino acids, also known as subunits or residues. The knowledge about the organization of these series is rare; the flat file format of these databases hides semantics of data, and the relationships/hierarchies are not clear, too. These databases do not support ad hoc and complicated queries, thus the data structure has many differences from traditional text.

From a basic and optimistic view, "a sequence is a mapping between a collection of similarly structured records and the positions of an ordering domain" [32]. Thus various sequences are just differently ordered domains and collections of record combinations. The mapping mechanism is shown in Fig. 1. We can say that the mapping method between DNA sequence and its database is in time domain and that between protein sequence and its database is in coordinate domain. Under this investigation, we can find that the basic and essential unit of the mapping is an "index," so it is possible for an index to be a base for a database that is efficient enough for homology search.

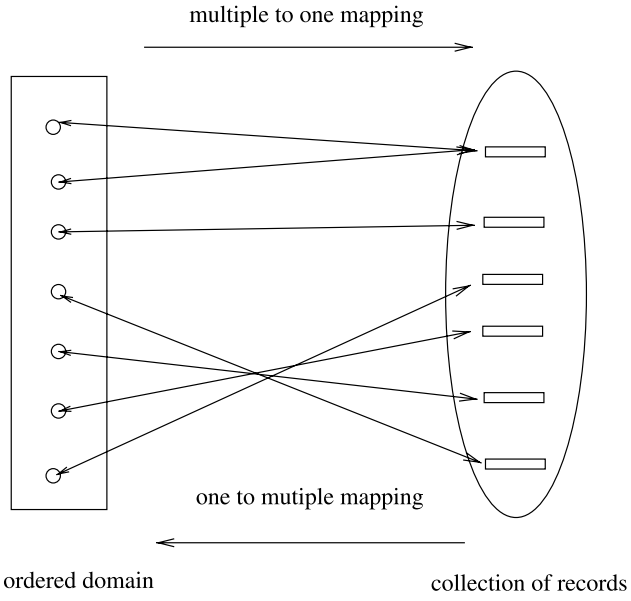

Fig. 1 A sequence mapping mechanism

2.2 Index necessity for genomic homology search and its advantages

An index for a genomic homology search is a "subsequence" extracted from genomic databases by a lexical analysis routine. Sometimes the "subsequence" is real subsequence of a sequence, while sometimes it is the one in a translated meaning, for example, a transformed one. The goal of an index is to minimize the storage requirements for the necessary access or retrieve and to minimize the cost of obtaining the original data relevant to the index. The cost includes the time to build an index, the time of filtration process and the space to store the index on disk. Tradeoffs between these requirements must often be taken.

For an exhaustive search method, caching is very important to reduce the access time to the genomic data. However, the cache system is too expensive and its capacity is very limited in current technology. The primary goal of caching is to reduce the amount of access and the access time. The index method is an efficient way to solve such kind of problems by cutting down the memory storage requirement and access time. Until finding other better methods, the index method is necessary for solving the serious problems faced by exhaustive search methods.

Until now, popular opinion on index based search algorithms is that despite the likely benefits of indexing in reducing query evaluation costs, existing indexed-based systems are not favored over exhaustive approaches because of limitations in index systems. However, index based search tools are true candidates for solving the problem for the following prominent advantages:

1. **Economical**: The index works as a filter in the homology search procedure, index based search algorithms only fetch a small number of sequences as likely answers

from the database, thus they are much faster, and use less computing resources. The cost of the same search will be greatly decreased;
2. **User independence**: The indices act as an intermediary between user and genomic databanks; it removes reliance on the external service, network delays, and the characteristics of the databanks, thus lets users be independent to the genomic databanks and the platform of servers;
3. **Convenience**: It is fully integrated with a database engine, the user can be transparent to retrieve one database with the indices, and so it is very convenient;
4. **High quality**: The index based algorithm is exhaustive instead of heuristics and can also allow approximate search. Its search time only depends on the index and does not need to depend on the databases;
5. **Flexibility**: It enables different statistics in sequence evaluation. That is to say, it can apply statistics based on the indices while it is not necessary to consider the genomic sequence.

2.3 Index construction method

Index based search algorithms are a very important part of genomic search methods, and how to construct indices is the key to an index based search algorithm. To construct an index, many lexical analysis schemes are possible and have been tried by experts. However, all the works start from a small viewpoint of the authors and are not guided by a total view of index models, such kinds of practical ideas make the index not as powerful as it should be. From a total analysis and view, we classified the index construction methods into three categories.

2.3.1 Length based construction method

In genomic databases, the sequences are linear, so it is intuitive to find some short subsequences (words) to construct the index, the length of the adopted subsequence can be either fixed or a variable. During the construction procedure, when a certain length and a certain alignment score are satisfied for a sequence segment, one item of an index is generated. The index can be constructed based on query sequences or object sequences, and ordering mechanism can also be applied to the index to improve its efficiency.

2.3.2 Special transformations based construction method

This method uses special transformations to construct an index. The special transformation can be of modern technologies such as wavelet, metric analysis, genomic statistics, and etc. For this method, the sequence should be changed into one kind of vectors (events) with a time, frequency, or another characteristic variable at first, then, based on the vectors, specific transformation is applied to the vectors to construct the index. The advantage of this kind of index is that it can prune most of non-desired sequences and reduce the real search problem to only a fraction of the original databases.

2.3.3 Mixed techniques based construction method

This stems from the above two kinds of indexing methods. So, in this method, the two above methods are mixed to generate the index.

Normally, there is an assumption that the genomic databases are static, thus the index is developed based on a fixed set of sequences. In a limited period and for a fixed genomic database, the index will stay consistent and will not need to be updated frequently.

In the following sections, basing on the analysis of the index methods used, we try to classify the popular index based homology search algorithms, and analyze and compare their characteristics.

3 Length based index algorithms

For the length based index ones, one can imagine distinctly there are two kinds: fixed length based index ones and variable length based index ones.

3.1 RAMdb

Rapid Access Motif database (RAMdb) is a algorithm used for finding short patterns in genomic databases [12]. In this system, each genomic sequence is indexed by its constituent overlapping intervals in a hash table structure. For each interval, an associated list of sequence numbers and offsets is stored. This allows a quick search of any sequence matching a query sequence.

RAMdb is best suited for the lookup of query sequences whose length is on the order of the indexed interval length. RAMdb has been shown to result in a up to a 800-fold speedup in search time over comparable exhaustive approximate pattern matching approaches.

Its limitations are multiple. RAMdb requires a large inverted index twice the size of the original flat-file database (including the textual descriptions), and suffers from lack of special-purpose ranking schemes designed for identifying initial match regions. In addition, the non-overlapping interval means false dismissals unless the frame happens to coincidentally align with the start of the interval frame.

3.2 FLASH

The FLASH search tool [6] redundantly indexes genomic data based on a probabilistic scheme. For each interval with length n, the FLASH search structure stores, in a hash-table, all possible similarly-ordered contiguous and non-contiguous subsequences with length m that begin with the first base in the interval, where $m < n$.

The hash-table stores every permuted m-length subsequence, the sequences containing the permuted subsequences, and the offsets within each sequence of the permuted subsequence. The key idea of the FLASH tool is that the permuted scheme gives an accurate model that approximates a reasonable number of insertions, deletions, and substitutions in genomic sequences.

The authors pronounced that FLASH was tens of times faster for a small test collection than BLAST [3, 4] and was superior in determining homologies accurately and sensitively in database searching.

FLASH has also great limitations. FLASH utilizes a redundant inverted index which is uncompressed, stored in a hash-table, and impractically large. For a nucleotide collection of around 100 Mb, the index requires 18 Gb on disk, around 180 times of the collection size. The 10 times performance is doubted to be attained on general purpose hardware unless the collection is sufficiently small such that swapping the index in and out of memory is not a serious problem.

3.3 Variable-length interval based algorithm

Twee-Hee Ong et al. [26] proposed a filter-and-refine approach to speed up homology search processes. It has some new aspects:

- The approach can handle motifs (one kind of index) of variable length.
- Inexact match is introduced, this can improve the effectiveness.
- Motifs are represented by a m-bit *signature* vector, where m can be chosen depending on the trade offs between performance and storage. The set of all signatures corresponding to the database sequences builds the signature file.

The filter-and-refine process extracts motifs from the query sequence(s) and generates a query signature at first, then the query signature is matched against the signature file to return a candidate set of sequences.

The method use one efficient and compact strategy to represent the motifs of the database sequences. The procedure is as follows: in order to map each database sequence to a V-bit signature vector, for a sequence with k motifs, $m_1, \ldots, m_k$, initially, all the V bits of the signature are set to 0 and a hash function $h(m)$ is defined to map a motif m to a value in the range 1 to V. Then, for motif m_i, where $1 \leq i \leq k$, bit $h(m_i)$ is set to 1. If there is a collision case for $m_i \neq m_j$ such that $h(m_i) = h(m_j)$, false drops occurs.

Two sequences are similar if they share some common motifs. With the signature representation, a simple logical "AND" operation is used to compute the intersection between two signatures (two sequences). The similarity measure *SIM* between query sequence signature Q and database sequence signature D is calculated as follows:

$$SIM(Q, D) = \frac{BitSet(Q \wedge D)}{BitSet(D)}, \tag{1}$$

where $BitSet(BS)$ denotes the number of bits in the vector BS, and $\wedge$ represents the bitwise logical-AND operation.

If both sequences share many common motifs, the similarity computed will be close to 1. These values can be ranked to form the candidate set by the top ranking sequences.

The process is really quick because it is a bit vector comparison. In the refinement phase, query sequence and candidate sequences are matched directly by exhaustive methods such as FASTA [29] or BLAST to pick out the answer set.

The author adopted the entire protein sequence database PIR1-PIR4 that have about 190874 protein sequences to test the algorithm. With $l_{\min}$ set to 3 and 4, and

minimum support set to 4.0% and 0.9% respectively, a total of 3322 and 3326 motifs were generated. When scanning only 20% of the database sequences, less than 10% of the good answers are missed by this scheme when compared to FASTA, and the missed answers rank fairly low under FASTA's answers.

As for the retrieval efficiency, for long sequence, for example, 600*bp* query sequence, it gains 50% more than that of FASTA. For short sequences, the gain is less significant.

The authors concluded that:

- For test on real data set with reasonable size m, the algorithm has good performance.
- The sensitivity of the algorithm is comparable to FASTA and the computation of the algorithm is faster.

The doubtful part is the result. We found the length of chosen query is around several hundreds, though this is a normal state, it is not enough to predict that the algorithm is more efficient for long query sequences. More experiments are necessary to argue the viewpoint.

Similar multi-step methods, filter-and-refine methods, are detailed in other papers [19, 21]. Some characteristics used in these papers may be borrowed to sequence alignment, but the structure of GenBank is a limitation.

Chattaraj [8] provided another variable length interval based approach for homology search. The authors showed that one can achieve a balance between the speed and accuracy of fixed length choices. The method can be integrated with other algorithms as a preprocessing procedure, at least it can work with CAFE.

3.4 BLAT

BLAT (BLAST-Like Alignment Tool) [20] is similar in a lot of ways to BLAST. The program rapidly scans for relatively short matches (hits), and extends these into high-scoring pairs (HSPs). However, BLAT differs from BLAST in some significant ways. BLAST builds an index of the query sequence and then scans linearly through the database. However, BLAT first builds an index of the database and then scans linearly through the query sequence. BLAST triggers an extension when one or two hits occur in proximity to each other, while BLAT can trigger extensions on any number of perfect or near-perfect hits. BLAST returns each homology area between two sequences as separate alignments, while BLAT stitches them together into a larger alignment.

BLAST delivers a list of exons sorted by exon size, with alignments extending slightly beyond the edge of each exon. BLAT has special code to handle introns in RNA/DNA alignments. BLAT effectively unsplices mRNA onto the genome by a single alignment that uses each base of the mRNA only once and correctly positions splice sites.

BLAT uses a simple and reasonably effective search stage to look for subsequences of a size k which are shared by the query sequence and the database.

BLAT can be divided into 4 steps:

- To find an initial match;
- To clump hits and identify homologous regions;

- To search for near perfect matches;
- To execute detailed alignment (maybe continued by a stitching and filling in procedure).

BLAT can handle very long database sequences efficiently, but it is more efficient when used for a short query sequence than for a long query sequence. It is not recommended when the query sequence is longer than 200000 bases.

BLAT is more accurate and 500 times faster than popular existing tools for mRNA/DNA alignments. It is 50 times faster for protein alignments at typical sensitivity settings when comparing vertebrate sequences. BLAT's speed stems from an index of all non-overlapping K-mers in the genome.

Its index is small enough to be fit inside the RAM of inexpensive computers and needs only to be computed once for each genome assembly. For example, the BLAT only needs five bytes per index entry, while it is eight in the published version of SSAHA. Because BLAT only indexes nonoverlapping words, it can index the human genome at the nucleotide level in 0.9 GB and index a *RepeatMasker* masked and translated human genome in 2.5 GB. It takes 30 minutes to build the index for the human genome, and once it is ready, it can be used typically for hundreds or thousands of query sequences.

BLAT also indexes the database rather than the query sequence. This is more than anything responsible for the relatively high speed of BLAT comparing to BLASTX or other popular tools. Rather than having to linearly scan through a GB database of sequences to look for index matches, BLAT only has to scan through a relatively short query sequence. This is similar to SSAHA, but SSAHA does not implement unspicing logic and always uses a single perfect match as a seed.

3.5 Piers

Piers [7] is a method similar to PatternHunter for that it allows spans between piers. The pier p in the model is defined as a segment with length ℓ_p and has a location at position *pos* in a data sequence. Formally, a pier is defined as a tuple $< p, pos >$. The method is based on the observation that similar subsequences would have similar subsequences.

For the extraction of the piers, the method takes an assumption that users are only interested in high similarity region which is of length greater than a minimum length $\ell_{\min}$. The piers are extracted randomly from the data sequence based on the following principle: At least k piers should be contained in any subsequence with length no less than $\ell_{\min}$, i.e., $((k+1)\ell_p + k\ell_s) \leq \ell_{\min}$, where ℓ_s is the length of span between neighbor piers.

After extracted, the piers are stored in a hash table *HTable* to ensure efficient access.

The sequence similarity search algorithm based on the hash-based pier model consists of three steps: (1) generating the query pattern with size of ℓ_p from query sequence Q; (2) searching for pier candidates among the hashed piers; and (3) post-processing the candidates to concatenate adjacent candidates in order to form final alignments with a high alignment score.

Let pier set be P. The total space complexity of the Hash structure is $O(4^{\lambda} + 4^{2\omega} + \omega|P|)$, where $\omega = \ell_p - \lambda$ and λ is the length of Hashed prefix. Typically, the hash table size is small enough to be kept in the main memory of a computer.

For each query pattern q of the query Q, let the pier set of the neighbors for q be N, the time complexity of the query is $O(\alpha|Q||N|)$, with the loading factor $\alpha = |P|/4^{\lambda}$. Under symmetric cases, considering letter repetition in a sequence and the neighboring of query patterns, time complexity can decrease.

The experiments show that the algorithm's efficiency is higher than BLAST for a series of query processing. As for the preprocessing, because the model simply extracts piers and hashes them rather than processes each segment in the sequence database as BLAST, the efficiency is higher. For the query processing, the method outperforms BLAST $2 \sim 10$ times when the size of data sets varies for both groups of test queries adopted by the authors.

The sensitivity and accuracy for a pier model can be provided by theoretical analysis. For the two subsequences with length $|S| = |Q| = L$, if query Q and candidate subsequence C have $edit(Q, C) \leq \zeta$, and if the edit distance between the pier randomly picked in S and the corresponding alignment segment of Q is ζ', the probability for S to be found is:

$$P(\ell_p, L, \zeta', \zeta) = \frac{\sum_{i=0}^{\zeta'} \binom{\ell_p}{i} \binom{L - \ell_p}{\zeta - i}}{\binom{L}{\zeta}} \tag{2}$$

4 Transformation based index algorithms

Transformation based index algorithms are all based on special technique(s), and at the same time, these transformations combine properties of genomic data.

4.1 CAFE

CAFE [34–37] is a partition based search approach, where a coarse search using an inverted index is used to rank sequences by similarity to a query sequence, and a subsequent fine search is used to locally align only a database subset with the query.

In our opinion, this method can be extended to other algorithms. For example, by removing single-member queries and long queries, the query set could be reduced by a factor of around two.

The CAFE index consists of three components:

1. A search structure. The search structure contains the index terms or distinct intervals, that is, fixed-length overlapping subsequences from the collection being indexed.
2. Inverted lists. Inverted lists are a carefully compressed list of ordinal sequence numbers. Each list is an index of sequences containing a particular interval. The cafe inverted file indexing scheme is extended so that within each postings list is stored not only the ordinal sequence number that contains the interval, but also offset information. For example, consider the following postings

list *ACCC* 12, (3 : 144, 154, 962); 38, (2 : 47, 1045); . . . in which the indexed sequences, the 12th and 38th, contain the interval ACCC. The interval occurs 3 times in the 12th sequence, at offsets 144, 154, and 962, and twice in the 38th sequence, at offsets 47 and 1045.

3. A mapping table. Mapping tables map ordinal sequence numbers to the physical location of sequence data on disk.

Queries are evaluated by representing the query as a set of intervals, retrieving the list for each interval, and using a ranking structure to store a similarity score of each database sequence to the query. Similar to FLASH, CAFE uses an overlapping interval.

CAFE uses a compression scheme to make the index size more manageable. To reduce the retrieval overhead of using an index, compression techniques used for text database indices and string indexing are used to reduce index size. The benefits of compression are two-fold: there is a saving in space used by the index and often a saving in query evaluation time, if retrieval of compressed lists and subsequent decompression is faster than retrieving uncompressed lists.

The CAFE method consists of a coarse search and a fine search. The coarse search is based on an index and used to select candidate strings that have the potential to be good answers. The fine search is used to decide which candidate is the real one we want. The coarse search involves retrieval of the inverted lists corresponding to unstopped intervals in the query string. Those sequences that fall within the top m according to frequency of occurrence of query intervals are presented to a fine searching algorithm for ranking against each other. The CAFE method returns the best m sequences regardless of their similarity values for an arbitrary m.

As for the space complexity, being compared to the GENBANK108 and GENBANK97 collections, the CAFE indices are 2.2 times the size of the collections, while the corresponding VERTE index is 2.5 times the collection size, and the index for PIRSF is 1.9 times the collection size. If without incorporating the special-purpose CAFE compression scheme, the uncompressed fine-grain inverted list size will be about more than two times larger.

As to the retrieval effectiveness, the experiments show that CAFE searching is only marginally less accurate than BLAST1 and FASTA, and that BLAST2 is 1–2% lower in precision at low recall levels and has up to 10% lower precision at higher recall levels. Most importantly, CAFE searching has both similar underlying heuristics and performance to FASTA which is the most accurate rapid homology search system.

As to the search speed, searching with CAFE can be over eighty times faster than that with FASTA and eight times faster than that with BLAST2. The comparison is shown in Table 1.

The author notes that although it is more computationally efficient than the exhaustive methods, while "exhaustive systems generally have better retrieval effectiveness."

The authors also pointed out that applying popular filtering techniques unselectively to all queries may reduce retrieval effectiveness, and CAFE uses the overall motif distribution in a database and filters all queries to mask low complexity regions prior to searching, which is a common practice to improve the accuracy of rapid homology searching and the search speed. XNU, SEG, SIMPLE, SAPS, CENSOR and CAFE all similarly reduce the retrieval effectiveness.

Table 1 Mean elapsed time in seconds for 41 nucleotide queries

	BLAST 1	BLAST 2	CAFE	FASTA
GBMAM	0.6	0.4	1.2	8.0
VERTE	6.8	7.4	3.2	108.4
GENBANK97	67.1	19.2	9.2	823.0
GENBANK108	192.5	182.5	20.2	–

Unclear problems, for example, whether the low complexity region is popular, how many percents of the total and etc., should be dealt with by a good algorithm. FASTA can be used to assess the reliability of answers to a given query. This is done by the actual CAFE algorithm.

The advantage of using overall collection frequencies is that frequent motifs common to both homologous and unrelated sequences are filtered. The method is 14 fold speedup over BLAST. The disadvantage of the method is that distant sequence relationships are lost and retrieving effectiveness is lower.

The partition based index and retrieval is a good method shown by CAFE. Another paper based on this kind of index focuses on text retrieval [38].

4.2 PropSearch

The PropSearch tool [14] is proposed to detect the functional and structural homologies in the twilight zone of sequence similarity, i.e., when the sequence identity falls below about 25%, the sequence identity is a sequence structural relationship in case of convergent or far divergent evolution. The basic idea is to utilize the conserved properties in the similar structures for database searches. It uses a notion of sequence space similarity between the two extreme approaches of "strict sequential order" and "amino acid content irrespective of order."

The authors proposed to neglect the order of amino acids in a sequence and to represent the sequence as a vector of 144 different characteristics, notably residue frequency, molecular weight, average residue-size, average hydrophobicity, and average charge.

One important rule in the algorithm is that characteristics such as hydrophobicity that are known to be stronger indicators of homology, carry more weight than those are known to be lesser indicators.

A genetic algorithm optimizes the property weights, and the optimization is stopped at generation 50. To avoid artifacts from overtraining, database searches were performed using the weights at generation 30. The optimization is performed using 100 weight vectors, each vector representing the weights of 144 properties. All 100×144 property weights were initialized to 1.0 at generation 0.

The "fitness" of a gene was calculated by a four-step procedure: (1) Take the 1322nd sequence and calculate the distance between the query sequence and all other 1321 sequences; (2) Sort sequences on distance; (3) Calculate the average rank of family members by $R_{\text{fam}} = \sum (R_i - (i-1))/N$, where i labels the family member, R_i is the rank of family member i in the list of hits; N is the number of family members; (4) Do step (1) to (3) for all 1322 sequences. The fitness of a set of weights is

defined as the sum of 1322 R_{fam}. A low average rank or high fitness results when a query sequence collects its family members at the top of the output, separating them from members of other families with higher PropSearch distances. In each generation, the evaluation of fitness was done for 100 genes and the ten highest scoring genes were reproduced.

After gene reproduction, genes were mutated with a probability of 0.035 and recombined with a probability of 0.2, with the exception of the gene number 1, which was neither mutated nor recombined, but kept. To speed up the optimization, genes 30 to 50 were subjected to a five fold higher mutation rate.

Query sequences are translated into vectors, search results are returned as a ranked list of sequences in decreasing order of Euclidean distance from database sequence vectors.

The method is implemented in several steps:

- Calculate a property vector by 144 numerical values;
- Preprocess a database of properties, i.e. for each sequence in the database, SwissProt database for example, a property vector of 144 numerical values is calculated;
- Calculate the property distances as the root weighted mean square difference of the components of the property vectors (Weighted Euclidean distance) by $D = \sqrt{\sum(|A_i - B_i|)^2 W_i}$, where A_i is property i of protein A after normalization by database sigma; B_i is property i of protein B after normalization by database sigma; W_i is weight for property i;
- After all distances between the query vectors and each database vector are calculated, distances are sorted and high scoring proteins, i.e. those with a small distance relative to the query protein, can be inspected for potential structural and functional homology.

The authors pointed out that PropSearch is a useful protein database searching tool in the context of genome analysis in case for which conventional alignment tools find no sequences significantly similar to the query protein. The similar sequence sets identified by PropSearch and other conventional alignment tools are, in general, only partially identical, and while PropSearch, on the average, finds some known members of a particular protein family, it may fail to find others. On the other hand, PropSearch may find family members not detectable by other alignment tools, because no similarity at the level of sequential alignment is present. The tool is not meant to compete with, but to complement the conventional alignment tools.

The comparison of PropSearch, FASTA, BLAST for the capability to detect remote homologies are shown in Table 2.

In case 1 and case 2, the actual number of top ranking sequences examined is 200 and 1000, respectively. We can see from Table 2 that PropSearch is very sensitive. For

Table 2 The capability comparison of PropSearch, FASTA, BLAST to detect remote homologies

Algorithm	Variable	Detected sequences in case 1	Detected sequences in case 2
PropSearch	–	222	771
FASTA	–	134	350
BLAST	0	–	–

example, for 592 globin sequences, though the sequence alignment is not detectable by traditional algorithms, PropSearch found 12 pairs with distance below 12 (i.e. reliability about 70%) and 93 pairs with distance below 13 (i.e. reliability about 55%).

Because the index is based on the 144 properties, the index size is very small.

As for the speed, the original idea in the literature is to improve the sensitivity and no report on its speed comparison with other methods is provided. We judge it is a little faster than other popular methods with the small index and the index construction payload.

Other studies have found that experience with PropSearch suggests it is only reliable for detecting evolutionary close similarities between highly homologous sequences and that subsequent alignments are required to verify homology. Lack of positional information of physicochemical properties, as commonly used in the exhaustive schemes, is the likely contributing factor to its poor retrieval effectiveness. In addition, most, of the properties used by PropSearch are not shared by the nucleotides that generate the amino acid sequences. At present, there are no known set of properties which may be used to form the nucleotide vector.

4.3 Bitmap indexing structure

The authors [27] employ a bitmap indexing structure to condense and encode each data sequence into a shorter index sequence. During query processing, the bitmap index is used to filter out most of the irrelevant subsequences, and false positives are removed in the final refinement step. The attraction of the strategy is its capability of reducing response time substantially while incurring only a small space overhead. The difference between bitmap indexing structure (BIS) and any other structure is that BIS is compact and has a simple generation mechanism.

The algorithm has three main steps:

1. BIS construction procedure. In this step, a hash function $f : D \rightarrow N$ is defined first to map each element d in the database to an index element n. Then, the hash function f is applied to every sequence in the database to produce a bitmap index.
2. Filtering step. This is a common step in other algorithms.
3. Result analysis based on a cost model.

To study the effectiveness and efficiency of the processing strategy, a cost model is developed. The purpose of the model is three-fold:

1. When constructing an index for a sequence database, the model can provide the best setting for N_{bits}, the number of bits needed for each index element.
2. The model can help a database system to decide whether BIS will speed up a particular query.
3. The model can estimate the processing time if BIS is employed.

The cost model is more accurate than other models because it considers the I/O communication time and computation time.

The authors adopted response time and speedup to evaluate the algorithm's performance. The following effects are considered in the experiments.

- The number of bits per index element N_{bits} should be as small as possible, a smaller N_{bits} lowers filtering cost but raises R_{match}.

- For the effect of query length, the speedup rises roughly linearly with $\log_2 Q_{\text{Len}}$, and the response time for BIS gets reduced a lot due to fewer subsequences dominate rising filtering cost and it only rises a little with the longer query.
- As to the effect of data size, the response time increases linearly with the data size, and the achieved speedup remains constant.
- For the data distribution, if the skew of index distribution is between 20% and 80%, the performance of BIS remains stable, while for more skewed index distribution, BIS deteriorates rapidly and the filter should be reconstructed with an updated hash function to restore BIS's effectiveness.
- For the effect of "fixed length don't care" segments, response time of BIS remains almost unchanged except that when the number of data elements in the query strings becomes less than six, BIS is faster.
- For the effect of "variable length don't care" segments, the overall response time is the sum of the individual segment's processing time.
- For the effect of edit distance, the BIS is better for larger edit distances if the query string is longer.
- As to D_{bits}, it has little effect on either the efficiency of BIS and the accuracy of the cost models.

As an example, by constructing an index with $1/8$ the size of database, BIS gets five times speedup for query strings that are longer than 100 data elements.

4.4 Metric space indexing techniques

The authors [9, 10] investigated two different indexing techniques, namely the variations of GNAT trees and M-trees, to support fast query evaluation for local alignment, by transforming the alignment problem to a variant metric space neighborhood search problem.

The algorithm integrates a fully sensitive indexing technique for coarse search.

The standard neighborhood query supported by index structures for metric space searching is for the following problem (P): Given a point $x \in X$ and radius $r > 0$, find all $y \in Y$ s.t. $d(x, y) \leq r$.

For the alignment query, we need the index structure to support the query for the following problem $(P*)$: Let f be a function from X to the set of real numbers. Given a point $x \in X$ and a radius $r > 0$, find all $y \in Y$ s.t. $d(x, y) \leq f(y) + r$.

Based on the above mechanism, what we need to do first is to transform the sequence alignments into edit distance evaluation that we can utilize conveniently.

For the transformation of local alignments to an edit distance evaluation, the score of local alignment for two sequences x and y is defined at first by:

$$\begin{aligned} \mathit{local\_alignment\_score}(x, y) = {}_{\text{def}} \max\{c \times (\text{total length of alignment}) \\ - (\text{penalty of gaps})\}, \end{aligned} \tag{3}$$

where $c > 0$ is a predefined constant, and the gap penalty is a non-negative affine function with respect to the total length and the number of the gaps.

If there is a score-threshold σ, and let $d(x, y)$ denote the edit-distance between sequence x and sequence y, then we get

$$d(x, y) \leq \text{len}(x) + \text{len}(y) - 2\sigma/c. \tag{4}$$

For a given point x and score σ, we get the corresponding radius $r(x, \sigma) = \text{len}(x) - 2\sigma/c$. The above relation is equivalent to $d(x, y) \leq r(x, \sigma) + \text{len}(y)$ which is an instance of problem $(P*)$.

Two different indexing techniques used by the algorithm is as follows:

GNAT: At the top node of a GNAT, several distinct split points are chosen and the space is divided into Dirichlet domains based on those points. The remaining points are classified into groups depending on what Dirichlet domain they fall into. Each group is then structured recursively in the same manner.

M-tree: M-tree uses a different strategy to maintain the index structure with two properties:

- The tree is balanced for all paths having the same length;
- In an M-tree, each node n stores the radius $r(n)$ which satisfies $r(n) \leq d(\hat{n}, \hat{m})$, where m is any descendant of n. Especially, $r(root) = \infty$ and $r(n) = 0$ for any leaf n.

The construction of an M-tree is performed by means of a series of node insertions. The algorithm at first finds the leaf of the M-tree that accomodates the inserted object and then handles the overflow caused by the insertion.

Aghili [1, 2] integrated a textual data mining technique, Singular Value Decomposition (SVD) dimensionality reduction technique, as an efficient filtration on genomic data to leverage the cost and scalability of the approximate search process. His work is a good example using metric space indexing method.

The advantage of the algorithm is that with the transformation, the calculation of distance in the frequency domain is linear in time/space, which is much more efficient being compared to the calculation of the distance in the original string domain which is quadratic in time/space.

The disadvantage is that false positives are introduced by the transformation.

The authors have not provided experiments for the performance evaluation and comparison with other kinds of methods, thus the method needs to be deeply investigated.

4.5 IDC-based algorithm

This approach [22] extracts homology candidates based on the data structure of Incrementally Decreasing Cover (*IDC*) from genomic databases. The new concept of *seriate coverage* builds the base of this algorithm. The problem of searching homology candidates is transformed to a longest increasing subsequence (*LIS*) problem with range constraints.

Based on the problem transformation, the algorithm has a two-phase filtration:

- Annotate query sequence and construct the hit list;
- Find homology candidates.

The time complexity to construct the gram index is $O(|D|)$ and the space requirement is $O(|D| + |\sum|^{L_I}) = O(|D|)$ for $|\sum|^{L_I} \ll |D|$, where $|D|$ is the size of genomic database, $|\sum|$ is the number of alphabet and L_I is the length of index.

Let L_Q denote the length of query grams and let $Z = L_I - L_Q$ if $L_I > L_Q$, and 0 otherwise. U is the number of nodes in the longest query-related location list, and

$V = |\sum|^Z$. The time complexity of annotating the query sequence and constructing the hit list is $O(|Q|+|H|+|Q|UV\log(|Q|V)) = O(|H|\log(|Q|V))$ for $|Q| \ll |H|$, where $|H|$ denotes the length of hit list.

The space requirement is $O(|Q|V + |H|) = O(|H|)$ for $|Q| \ll |H|$. For finding the homology candidates, the overall time complexity is $O(|H|W(1 + \log W)) \simeq O(|H|W\log W)$ and the algorithm needs $O(W_Q) = O(W)$ space.

Another characteristic is that it is a lossless filtration algorithm with user-specified error and seriate coverage levels. The design of variable-length of query grams provides more flexibility in various applications.

The authors pointed out IDC based filter (*IDCF*) is more than three orders of magnitude faster than QUASAR, and more than 10000 times faster than the exhaustive search with a comparable sensitivity level.

5 Mixed techniques based index algorithms

Mixed techniques based index algorithms take different technique mentioned above at different step and at least they use two techniques, we here group them into the third type.

5.1 Indexing using wavelets

By the wavelet indexing method, substrings are mapped into an integer space with the help of wavelet coefficients. These coefficients are indexed into Multi Resolution String Index Structure (MRS) using Minimum Bounding Rectangles (MBRs). Frequency Distance (FD) is used to reduce search space in the method.

Aiming at the problem of the range queries and nearest neighbor queries, Kahveci [17] in 2001 proposed a method to map the data substrings into an integer space with the help of wavelet coefficients. With the help of MBR, the method can index the coefficients and the index can prune 50–95% of the database, which reduces the disk I/O operations and CPU time. Its advantage is that the typical size of MRS index structure ranges between 1–2% of the database size, thus the method is a potential solution for the long query problem because the whole index can be easily stored in main memory.

Based on the Multi Resolution String (MRS) index structure, Kahveci [18] presented an algorithm which constructs a boolean match table for a given query string and database string. The algorithm can efficiently address the alignment of large genome strings problem. In the algorithm, each entry of the match table corresponds to a query/database substring pair, and the match table size is negligible compared to that of a database (typically 0.1% of the database). Later, hash tables are constructed from the match table. Once the hash table of a string for a slice is constructed, the marked substrings of the other string are read sequentially and exactly matching substrings (i.e. seeds) of the prespecified size (i.e. 11) are found using this hash table. The seeds are then extended in both directions to find better matches. In the end, the results are reported in a descending score order. This technique is called MAtch table based Pruning (MAP).

The exact string matching algorithm based on wavelets consists of:

1. Building an index. In this step, the data sequences are scanned to build MBRs at first, then, MBRs are stored in a spatial indexing structure.
2. Given the query string, determining the MBRs in which it is likely to lie;
3. To do a false positive elimination by matching query Q to those portions of database D that lie in the identified MBRs.

Based on the index, an extension is implemented. The objective is to answer queries of various lengths efficiently, and an MRS is maintained to do this. This translates to a different R tree corresponding to each resolution (or window size). It may be noted that only an MRS corresponding to window sizes that are powers of two will be constructed. This allows one to most efficiently handle a range of query strings of yet unknown lengths.

The attractiveness of wavelet based index structures [31] lies in the index size generated which is a fraction of that of comparable methods such as suffix trees. The author provided the performance comparison between the method and *MUMer*. The primary performance metrics include indexing time, querying time, and index size, and all are better than *MUMer*'s.

Index size is inversely proportional to cluster size. This can be related by the empirical formula: $index\_size\_per\_resolution = 64 \times database\_size/cluster\_size$.

The number of searched MBRs are orders of magnitude different between using queries of length 2, 4 and 8 and using those of length 16 and upwards. For queries of length 16 and upwards, the number of searched MBRs has a non-monotonic change—first increasing and then decreasing.

As to the indexing time, although MUMer is able to match incremental query (IQ) at small database size (under 512 k), it loses out to IQ very quickly, starting at moderate database size.

Query time is evaluated by the authors for a fixed query string of length 2 KB chosen from the Elegans gene itself mixed from two different locations. The variable is the minimum MUM size. The result shows that MUMer outperforms IQ when the minimum match size is small. However, with a minimum match size of 100, IQ takes around 8 seconds. Given the minuscule size of its index, that is acceptable compared to MUMer. MUMer builds a 249 MB index for the gene. In contrast, IQ uses only 0.226 MB.

The advantages of the algorithm lie in: (1) incremental window shifts, and (2) limited window for duplicate elimination.

The method is similar to BLAST in quality, but up to 97 times faster than BLAST without decreasing the output quality.

There are also some drawbacks. The efficiency is not comparable with other methods, and the edit cost is too simple. The method is only based on edit distance, no general scoring schemes are adopted. Another drawback is that it is not suitable for average DNA/protein query lengths.

Wavelet theory has not been presented for homology search initially. Here, the algorithm adopts wavelet theory to code object sequence and query sequence, and the homology search is executed on the coefficients. So, from the view of wavelet theory, the index is also transformation based.

5.2 Indexing using suffix trees

Suffix trees [15, 16] are primarily used for exact matching, but can be adapted for similarity matching. However, the method has some issues: caching and check pointing.

A suffix tree indexing a string of length N has N leaf nodes, one per suffix (suffixes being numbered from 1 to N). Each edge is labeled with a non-empty substring, and at each branching node the starting letters of the outgoing edges are different, so each path from root to a leaf spells the suffix that starts at the sequence position held in the leaf.

Suffix trees are compressed digital tries. Given a string, all suffixes are indexed. For example, for a string of length 10, all substrings that starting at index 0 through 9 and finishing at index 9 will be indexed. The root of the tree is the entry point, and the starting index for each suffix is stored in a tree leaf. Each suffix can be uniquely traced from the root to the corresponding leaf. Concatenating all characters along the path from the root to a leaf will produce the text of the suffix.

To change a trie into a suffix tree, each node which has only one child is conceptually merged with that child, and recursively, the nodes are annotated with the indices of the start and end positions of a substring indexed by that node. Commonly, a special terminator character is also added to ensure a one-to-one relationship between suffixes and leaves. Otherwise, a suffix that is a proper prefix of another suffix would not be represented by a leaf. The change from a trie to a suffix tree reduces the storage requirement from $O(n^2)$ to $O(n)$, where n is the sequence length.

The new incremental construction algorithm trades ideal $O(n)$ performance for locality of access on the basis of two decisions:

- To abandon the use of suffix links;
- To perform multiple passes over the sequence, as well as constructing the suffix tree for a subrange of suffixes at each pass.

If to encode the tree without making each node an object, one would require 12 B per node, then one needs around 21 B for each indexed character. Further compression could be obtained by using techniques similar to those proposed by Kurtz. Its space is $O(n)$, this is a smaller size, compared to others.

Time consumption is on average $O(n \log n)$, and the worst case is $O(n^2)$.

The suffix tree can be suitable for large sequence, i.e., the suffix tree is a persistent method for large sequence.

The authors stated that their algorithm is scalable and can be adjusted to run on computers with different memory characteristics. But more work is required to optimize the tree building, and to investigate the object placement on disk and as well as its influence on query performance. The construction procedure parallelization needs to be studied too.

The main problems of the suffix tree approach are twofold: (a) Suffix trees are inefficient at managing mismatches. This approach is perfect for highly similar sequences but fails to recognize more distant homologies; (b) Suffix trees have a large space overhead, and the best practical implementations still require about 9 times the database size and do not handle secondary memory well. Tree compression, alternative data structures, and data clustering are useful for the improvement of using suffix tree indexing technique.

Suffix array is a weak version of suffix trees [23]. It requires much less space (about 4 times the text size), but it has a mall penalty ($O(\log n)$ in time penalty factor) over search time.

In this method, we should construct the suffix tree based on the object sequence and query sequence first, and then, execute the algorithm on the suffix trees, while not on the object sequence and query sequence directly, thus the algorithm is also based on transformation.

5.3 iBLAST

iBLAST [11] is proposed as an indexed version of BLAST. The authors gave an initial implementation of the method, which uses a sequence-based index to catalog genomic databases in an NCR Teradata relational database management system (RDBMS). The Teradata utilizes parallel processing to achieve fast and accurate answers to queries.

In the method, the initial index based on sequences is comprised of a 16-mer word from the genomic database and a pointer to the location in the flat file where that "word" occurs. The index is built by the following method:

- The 16-mer word is converted into an integer for space saving. The conversion mechanism used the following scheme: A = 00, C = 01, G = 10, T = 11. A 32-bit binary number is generated to represent each 16-mer, which, in turn, is converted into an integer. For example, when a 4-mer word AGCA is located at position 1144 in the database, the word is encoded as 00100100, which is equal to the decimal integer 36, and its record in the database will appear as (36,1144);
- Create a unique primary index using the word and location.

In order to reduce space and maintain speed, a primary index was created on the word field and no unique index was employed. A unique primary index can facilitate the data to be more evenly distributed across the access module processors of the Teradata system. Due to the nature of genomic data, the skew among the processors is very low, so the unique primary index can be removed without too much sacrifice.

A linear relationship exists between the original database size and the Teradata RDBMS size. The Teradata RDBMS size is approximately 22 times larger than the database size due to the overhead of the index; this is a serious problem. The authors also noticed inconsistent behavior in smaller query sizes. A secondary index is introduced to alleviate the problem. At most, it is 100 times faster than queries performed without a secondary index. However, though query speed increased greatly, and the spikes and the inconsistent behavior from smaller query sizes are removed, the addition of a secondary index doubled the size of the database on the RDBMS.

The speed comparison of iBLAST to a sequential search tool, i.e. standalone BLAST shows that BLAST performs linearly as query size increases and iBLAST is virtually constant for a variety of query sizes. For the query time of entire ecoli genome comparing to the entire yeast genome, iBLAST performs 68 times faster than standalone BLAST.

About the sensitivity, the author did not provide a report, and it is left for the future version.

The authors' work is primary, and their proposal for accuracy, repeatability, speed test, using biological indices and integrating the relationships between genes are expected to verify the iBLAST algorithm. In the verification, one can take measures such as variable interval lengths, number of intervals, input sequence size, data type and etc.

In this method, the index is constructed based on a 16-mer word integer coding method, and the index is also based on a relational database management system. Clearly, it is a transformation based technique from this point of view.

5.4 SSAHA

SSAHA (Sequence Search and Alignment by Hashing Algorithm) [24] is based on organizing the DNA database into a hash table data structure (an index based on database), and the fact that computers with sufficient RAM to allow us to store a hash table that describes a database containing multiple gigabases of DNA, are available.

The authors used $w_j(S)$ to denote k tuple of sequence S that has offset j. The *offset* is the position of the first base of k-tuple with respect to the first base of S. The authors also adopted a two-binary digits encoding system for the 4 nucleotides.

The first stage of the algorithm is to convert subject sequences D into a hash table. The hash table is stored in memory as two data structures: a list of positions L and an array A of pointers into L. The hash table is constructed by making two passes through the subject data. On each pass, only the non-overlapping k-tuples in the subject sequences are considered. On the first pass, all non-overlapping occurrences in D of each of the 4^k possible k-tuples are counted to calculate the pointer positions for A and allocate the correct amount of memory for L. All words that have a frequency of occurrence that exceeds a threshold N is ignored to reduce the size of the hash table as well as effectively filter out the spurious matches attributable to repetitive DNA. After A is constructed, a second pass is implemented using A to place the position information into L at the correct positions.

For the search for occurrences of a query sequence Q within the subject database, it is processed base by base along Q from base 0 to $n-k$, where n is the length of Q.

For base t, a list of r positions of the occurrences of the k-tuple $w_t(Q)$ is obtained, and they are pointed to by entry $E(w_t(Q))$ of A. The list of positions is:$(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)$. Then the list of hits is computed:$H_1 = (i_1, j_1 - t, j_1)$, $H_2 = (i_2, j_2 - t, j_2), \ldots, H_r = (i_r, j_r - t, j_r)$. The list of hits is added to a master hit list M that is accumulated as t runs from 0 to $n-k$. From the leftmost to right most, the elements of a hit are referred to as the *index, shift*, and *offset*.

After M is built, it is sorted first by index and then by shift. Finally, scanning through M is used to look for hits for which the index and shift are identical. Each such hit represents a run of k bases that are consistent with a particular alignment between the query sequence and a particular subject sequence. The shift between consecutive hits in a run is allowed to be inserted or deleted by a small number of bases. With the sorting by offset, the regions of exact match between two sequences can be determined. Based on these, and by joining exact regions that are sufficiently close to one another, one can get the gapped matches.

The method will find matches only in the forward direction. For the reverse direction, it can do that easily by taking the reverse complement of Q and repeating the procedure.

Storage of the hash table requires $4^{k+1} + 8W$ bytes of RAM in total, in which the first and second terms account for the memory requirements of A and L, respectively, and W is the number of k-tuples in the database. For the storage of query sequences, it can be kept to a minimum by loading in query sequences from a disk in batches and using a 2-bit per base encoding method, normally it will add about 10–20% to the total RAM usage.

SSAHA is fast for large databases because the database is hashed; search time is independent of the database size as long as k is selected to keep $W/4^k$ small. Both FASTA and BLAST hash the query sequence and scan the database; therefore, the search time is related directly to the database size. The tradeoff is that SSAHA requires large amounts of RAM to keep A and L in memory.

SSAHA algorithm will, under no circumstances, detect a match of less than k consecutive matching base pairs between query and subject. $2k - 1$ consecutive matching bases are needed to guarantee that the algorithm will register a hit at a point in the matching region. With default settings, FASTA, BLAST, and MegaBLAST require at least 6, 12, and 30 base pairs, respectively, to register a match.

SSAHA is more effective for the sequence alignment of the same organism, but it always uses a single perfect match as a seed and does not adopt unsplicing logic.

SSAHA is based on a hash table and also adopts the two-binary digits encoding system for the four nucleotides, thus it is similar to iBLAST tool as a transformation based technique.

6 Conclusion and discussion

6.1 Performance comparison of the index based algorithms

Based on the above review, we show the performance comparison of the index based homology search algorithms in Table 3.

From the comparison, we can see that all index based algorithms are faster than classical heuristic algorithms. When it is based on transformations to construct the index, the index size will be smaller than the database size. For a length based construction method, the index size is larger than the database size. In some cases, it is too large for execution on long sequence.

6.2 Other applicable index methods

Heuristic algorithms [30] and the indexing scheme are two very different directions for homology search, although the indexing scheme is sometimes combined into heuristic algorithm. Heuristic algorithms put time efficiency over the cost of sensitivity, while index based algorithms try to get time efficiency with high sensitivity. There are some other techniques used to reduce search time for other types of databases. It is possible for these techniques to be applied on genomic databases and expand the successful index based homology search algorithms.

Table 3 Performance comparison of the index based algorithms

Algorithms	Index size	Speed	Sensitivity
RAMdb	2 times	~800	unavailable
FLASH	180 times	~10	more
Variable-length	a small part	faster than FASTA	comparable
BLAT	30%	500 times for mRNA/DNA,50 times for protein	comparable
Piers	small enough	2 ~ 10 times faster than BLAST	better
CAFE	~2 times	80 times of FASTA and 8 times of BLAST	a little less
PropSearch	very small	unavailable	better
Bitmap	a small part	5 times	comparable
Metric method	linear in space	linear in search time	a little less
IDC	very small	10000 times faster than exhaustive method	comparable
Wavelet	1–2%	97 times faster than BLAST	comparable
Suffix	linear	average $O(n \log n)$	comparable
SSAHA	$4^{k+1} + 8W$	average $O(n \log n)$	high
iBLAST	about 20 times	68 times faster than BLAST	unavailable

6.2.1 Feature vectors

Feature vectors are commonly used to compare similarity between query items and a search set. The goal of a feature vector is to extract features from the database objects and the query.

The only feature in DNA sequences is that nucleotides make up the patterns, and the patterns form sequences. Feature vectors consisting of counts of individual nucleotides, or nucleotide sequences, are not very useful for the following reasons:

- One sequence may be a subsequence of the other.
- Order of nucleotides is important and not accounted for by feature vectors.
- Traditional feature vectors only count exact matches.

The efforts in metric space indexing methods can promote the potential application ability of the feature vector.

6.2.2 Frequency domain mapping

Another technique used to reduce sequence dimensionality is to transform time domain sequences into the frequency domain [5]. Discrete Fourier Transform (DFT) can be used to transform a time sequence into frequency domain. The dimensionality of the transformed sequences is then reduced by throwing out some coefficients of the resulting transformation.

A DNA sequence may be viewed as a sequence in the time domain, in which each nucleotide is considered to occur later in time than the nucleotide to the left and earlier than the nucleotide to the right. Such a view allows a DNA sequence to be transformed into the frequency domain. However, since nucleotides may appear in any order with equal probability, sometimes the sequences will appear as white noise in the frequency domain.

Aghili [1] used a frequency transformation based method to filter the sequence to get a smaller index. The transformation method can be easily integrated with other heuristic algorithms such as BLAST, PatternHunter, and FASTA.

Based on the frequency transformation, the *Parseval Theorem* keeps frequency distance at less than or equal to edit distance. This property is the main driving force behind using frequency transformation.

Another advantage of frequency mapping is that the calculation of frequency distance is much more *time/space-efficient* compared to that of edit distance.

Ozturk [28] also proposed an effective method to transfer subsequences into numerical vectors and built efficient index structures on the transformed vectors. The transformation is similar to that T. Kahveci proposed. The authors did experiments on the transformation based distance functions and compared their (a) approximation quality for k-Nearest Neighbor ($k-NN$) queries, (b) pruning ability and (c) approximation quality for ε range queries. They found that distances $FD2$ and $WD2$ (i.e. Frequency and Wavelet Distance functions for 2-grams) perform significantly better than the others. The performance evaluation verifies that the frequency transformation based index structure is effective and promising. However, there is no currently completed performance evaluation based on true biosequence data.

6.2.3 Precomputation

Both FASTA and BLAST use a comparison of k-tuples as an initial step in the search process. One kind of precomputations is to generate feature vectors, which can indicate the presence or absence of high scoring k-tuples. Though sizable, such a feature vector would allow both FASTA and BLAST to quickly perform their first phase calculations. FASTA would be able to identify sequences containing hot spots and BLAST would be able to identify "seeds."

Though precomputation offers potential for a great time savings in the initial phases of FASTA and BLAST, there are drawbacks as well. Precomputed feature vectors are too large (4069 dimensions for a 6-tuple vector). To avoid large feature vectors, it is possible to precompute a lexicon of 6-tuple referencing the locations they occur in the database. For large databases, the lexicon would be smaller than the feature vectors. However, the method still suffers from an unreasonable inflexibility. The tuple size and the scoring matrix for a precomputed method are constant, and cannot be changed at search time.

6.3 Future directions

The requirements for homology or nearest neighbor search of nucleotide databases make it substantially different from other common search problems. However, searching for DNA sequences does subject to conventional and newly developed techniques to improve performance of database searching.

The following directions have been prominent in the index based homology search area. They are still active and have a huge impact on future research on index based homology search algorithms.

1. **Index schemes**: Index schemes have been experimented on with great success, but they are either limited by the types of queries on which they perform well,

or by the amount of space required by the index. One enhancement way is to incorporate a highly efficient index compression or to invent new indexing methods that provide a high mapping efficiency. We also notice that some successful methods adopt various transformation techniques to build and apply the index, and most of the time the index is adopted as a preprocessing for the genomic heuristic searching algorithms. If the index can be small and efficient enough, they will be used generally and will even be used independently to solve the homology search problem.

2. **Dedicated system**: Dedicated system [13, 25] combining several techniques are also tried. There are too many announced cases which second generation architectures have not produced because of the high production costs involved, thus a low cost system would be attractive. Furthermore, efficient parallel indexing algorithm will also promote the dedicated system development. A review of this topic is outside of the scope of this paper, for we are mainly paying attention to algorithms.
3. **Parallel and distributed implementation**: This is the most basic and intuitive method adopted both by software and hardware [33]. It is also true for index based homology search algorithms. Both high-level parallelization and fine grain-level parallelization are important, and hardware will prefer much to the fine grain-level parallelization to improve performance.
4. **Combined techniques**: We notice that there is no single index technique that can accomplish an action for the genomic search problem, and most of the time, various technique are combined or used in different steps (almost all fast alignment programs have two stages, while multiple stages are possible) to solve the problem. Thus, trying more combinatory index techniques is a promising prospect.

Newly proposed indexing and other techniques seem good, but require thorough live testing, including theoretical as well as practical study, in the following years/decades. Technology is not saturated yet, it is still open for us to make new contributions, so we have chance to get a new index method.

We can see that many index techniques in literature are similar to a certain extent, most of them are tested in software under the limitation of currently available computer systems, and few hardware systems have been studied recently. If we can consider hardware development, for example, hardware parallelization, new techniques such as reconfigurable computing [13, 25], and etc., and tune the algorithms to the hardware, there should be a new path to take.

Acknowledgement The work is partially supported by NSFC Grant 60403025 and PRA SI04-04. The authors thank Professor Dominique Lavenier for his valuable suggestions on this paper, and also thank anonymous reviewers for their comments to improve the readability of this paper.

References

1. Aghili SA, Agrawal D, El Abbadi A (2003) Filtration of string proximity search via transformation. In: Third IEEE symposium on bioinformatics and bioengineering (BIBE'03), Bethesda, MD, USA, 2003
2. Aghili SA, Sahin OD, Agrawal D, El Abbadi A (2004) Efficient filtration of sequence similarity search through singular value decomposition. In: Fourth IEEE symposium on bioinformatics and bioengineering (BIBE'04), Taichung, Taiwan, 2004

3. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. National Center for Biotechnology Information, National Library of Medicine, National Institutes of Health, Bethesda, MD
4. Altschul SF, Madden T, Alejandro A, Schaffer A, Zhang J, Zhang Z, Miller W, Lipman DJ (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. National Center for Biotechnology Information, National Library of Medicine, National Institutes of Health, Bethesda, MD, July 1997
5. Argyros T, Ermopoulos C (2003) Efficient subsequence matching in time series databases under time and amplitude transformations. In: ICDM, 2003, pp 481–484
6. Califano A, Rigoutsos I (1993) FLASH: a fast look-up algorithm for string homology. In: International conference on intelligent systems for molecular biology, Bethesda, MD, pp 56–64
7. Cao X, Li SC, Ooi BC, Tung AKH (2004) Piers: an efficient model for similarity search in DNA sequence databases. Sigmod Record, Special Issue
8. Chattaraj A, Williams HE (2004) Variable-length intervals in homology search. In: Asia-pacific bioinformatics conference, Dunedin, New Zealand, 2004
9. Chen W, Aberer K (1997) Efficient querying on genomic databases by using metric space indexing techniques. Technical Report No. 1056, German National Research Center for Information Technology
10. Chen W, Aberer K (1997) Efficient querying on genomic databases by using metric space indexing techniques. In: Eighth international conference and workshop on database and expert-systems applications (DEXA'97), Toulouse, France
11. Cooper G, Raymer M, Doom T, Krane D, Futamur N (2004) Indexing genomic databases. In: Fourth IEEE symposium on bioinformatics and bioengineering (BIBE'04), Taichung, Taiwan, 2004
12. Fondrat C, Dessen P (1995) A Rapid access motif database (RAMdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks. Comput Appl Biosci 11(3):273–279
13. Gardner-Stephen P, Knowles G (2003) A novel architecture for genomic sequence searching and alignment. In: Asia-pacific computer systems architecture conference, pp 180–192
14. Hobohm U, Sander C (1995) A sequence property approach to searching protein databases. J Molec Biol 251:390–399
15. Hunt E, Atkinson MP, Irving RW (2001) A database index to large biological sequences. In: Proceedings of the 27th VLDB conference, Roma, Italy, 2001
16. Hunt E, Atkinson MP, Irving RW (2002) Database indexing for large DNA and protein sequence collections. VLDB J 11(3):256–271
17. Kahveci T, Singh AK (2001) An efficient index structure for string databases. In: Proceedings of the 37th VLDB conference, Roma, Italy, 2001
18. Kahveci T, Singh AK (2003) MAP: searching large genome databases. In: Pacific symposium on biocomputing, Hawai, 2003
19. Kailing K, Kriegel H-P, Schonauer S, Seidl T (2004) Efficient similarity search for hierarchical data in large databases. In: Proc 9th int conf on extending database technology (EDBT 2004), Heraklion, Greece, pp 676–693
20. Kent WJ (2002) BLAT: the BLAST-like alignment too. Genom Res 12(4)
21. Kriegel H-P, Schonauer S (2003) Similarity search in structured data. In: Proc 5th int conf on data warehousing and knowledge discovery (DaWaK'03), Prague, Czech Republic, Lecture notes in computer science (LNCS), vol 2737, 2003, pp 309–319
22. Lee HP, Tsai YT, Sheu TF, Tang CT (2004) An IDC-based algorithm for efficient homology filtration with guaranteed seriate coverage. In: Fourth IEEE symposium on bioinformatics and bioengineering (BIBE'04), Taichung, Taiwan, 2004
23. Navarro G, Baeza-Yates R, Sutinen E, Tarhio J (2001) Indexing methods for approximate string matching. IEEE Data Eng Bul 24(4)
24. Ning Z, Cox AJ, Mulikin JC (2001) A fast search method for large DNA databases. Genom Res 11(10)
25. Oliver T, Schmidt B (2004) High performance biosequence database scanning on reconfigurable platforms. In: IPDPS04 (HiCOMB), Santa Fe, NM, IEEE, 2004
26. Ong TH, Tan KL, Wang H (2002) Indexing genomic databases for fast homology searching. In: Proceedings of the 13th international conference on database and expert systems applications, September 2002, Aix-en-Provence, France, pp 871–880
27. Ooi BC, Pang HH, Wang H, Wong L, Yu C (2002) Fast filter-and-refine algorithms for subsequence selection. In: Proceedings of the 6th international database engineering and applications symposium (IDEAS'02), Edmonton, Canada, July 2002, pp 243–254

28. Ozturk O, Ferhatosmanoglu H (2003) Effective indexing and filtering for similarity search in large biosequence databases. In: 3rd IEEE international symposium on bioinformatics and bioengineering (BIBE 2003), Bethesda, MD, USA
29. Pearson WR, Lipman DJ (1988) Improved tools for biological sequence comparison. Proc Natl Acad Sci USA 85:2444–2448
30. Rognes T, Seeberg E (1998) SALSA: improved protein database searching by a new algorithm for assembly of sequence fragments into gapped alignments. Bioinf 14(10):839–845
31. Roy A, Mullick A, Genomic indexing using wavelets. Available at: http://people.csa.iisc.ernet.in/~aroy/gene.doc
32. Seshadri P, Livny M, et al (1996) The design and implementation of a sequence database system. In: Proc of the 22nd VLDB conf, Mumbai, India
33. Shamir R (1998) Algorithms for molecular biology, Lecture 3. Tel Aviv University, Fall 1998
34. Willams HE (1997) Fast ranking strategies for genomic databases
35. Williams HE (1999) Effective query filtering for fast homology searching. In: Pacific symposium on biocomputing, Hawaii, pp 214–225
36. Williams H, Zobel J (1996) Indexing nucleotide databases for fast query evaluation. In: Proc of the 5th international conference on extending database technology, Avignon, France, pp 275–288
37. Williams H, Zobel J (2002) Indexing and retrieval for genomic databases. IEEE Trans Knowl Data Eng 14(1):63–78
38. Yang Y, Liu B, Zhang Z (2003) Partition based hierarchical index for text retrieval. In: WAIM, 2003, pp 161–172

Xianyang Jiang received the B.S. degree in safety engineering from Shenyang Institute of Aeronautic Engineering, Shenyang, China, in 1995, the M.S. degree in physical electronics and optoelectronics, and the Ph.D. degree in pattern recognition and intelligent system from Huazhong University of Science and Technology, China, in 1998 and 2003, respectively. Since 2005, he has been an Assistant Professor with Institute of Computing Technology, CAS. From 2004 to 2005, he was a postdoc fellow in INRIA, France. His current research interests include computer architecture, VLSI design, reconfigurable computing, and bioinformatics.

Peiheng Zhang received the B.S. degree in electronic engineering from Nankai University, China, in 1989. Now he is a Professor with Institute of Computing Technology, CAS. His main research interests include computer architecture, hardware design, and reconfigurable computing.

Xinchun Liu received the Ph.D. degree from Institute of Electronics, CAS, in 2000. He is an Associate Professor with Institute of Computing Technology, CAS. His current research interests include reconfigurable computing and high performance interconnection networks.

Stephen S.-T. Yau received the M.S. and Ph.D. degrees from the State University of New York at Stony Brook in 1974 and 1976, respectively. In 1976–1977, he was a member of the Institute for Advanced Study at Princeton. From 1977–1980, he was a Benjamin Pierce Assistant Professor at Harvard University. He received the Sloan Fellowship from 1980 to 1982. In 1980 he joined the Department of Mathematics, Statistics, and Computer Science, University of Illinois at Chicago (UIC) as Associate Professor. He was promoted to Professor at UIC in 1984. He has also held several visiting professorship positions at Princeton University (1981), Institute for Advanced Study (1981–1982), University of Southern California (1983–1984), Yale University (1984–1985), Institute Mittag-Leffler, Sweden (1987), The Johns Hopkins University (1989–1990), University of Pisa, Italy (1990). He was awarded the University Scholar (1987–1990) by University of Illinois at Chicago. In 2000, he received Guggenheim award. In 2002, he was awarded the IEEE Fellow. In 2005, he received the Distinguished Professorship award from University of Illinois at Chicago.

Dr. Yau has been the managing editor of Journal of Algebraic Geometry since 1990. He has been the Director of the Control and Information Laboratory since 1993. He has been the Editors-in-Chief of Communications in Information and Systems since 2000.