DSIM: A Distance-based Indexing Method for Genomic Sequences

Xia Cao[1] Beng Chin Ooi[1] Hwee Hwa Pang[2] Kian-Lee Tan[1]
Anthony K.H. Tung[1]
[1]Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
{caoxia,ooibc,tankl,atung}@comp.nus.edu.sg

[2]Institute of Infocommunication Research
21 Heng Mui Keng Terrace
Singapore 119613
hhpang@i2r.a-star.edu.sg

Abstract

In this paper, we propose a Distance-based Sequence Indexing Method (DSIM) for indexing and searching genome databases. Borrowing the idea of video compression, we compress the genomic sequence database around a set of automatically selected reference words, formed from high-frequency data substrings and substrings in past queries. The compression captures the distance of each non-reference word in the database to some reference word. At runtime, a query is processed by comparing its substrings with the compressed data strings, through their distances to the reference words. We also propose an efficient scheme to incrementally update the reference words and the compressed data sequences as more data sequences are added and new queries come along. Extensive experiments on a human genome database with 2.62GB of DNA sequence letters show that the new algorithm achieves significantly faster response time than BLAST, while maintaining comparable accuracy.

1 Introduction

Molecular biologists frequently query genomic databases to identify sequence homology. Such an operation involves matching a query sequence against each of the database sequences where the similarity comparison between two sequences typically has a complexity that is either linear or quadratic to the length of the sequences.

BLAST [4] is one such tool that is being used widely. For every pair of sequences to be matched, it looks for short fixed-length word pairs, or seeds, that are shared by both sequences and then extends the seeds to higher-scoring regions. While the approach works well with small databases, its performance deteriorates quickly as the size of genome database grows. Thus BLAST is not a satisfactory solution because not only are genomic databases exploding in size, the number of queries directed at them also has surpassed 40,000 queries per day [16] and is still rising. After "dissecting" the tool to analyze its behavior, we discovered that the seed searching step dominates(on average constituting more than 80% of) the response time for large databases. This prompted us to look for an efficient alternative to eliminate the bottleneck.

In this paper, we propose a new search algorithm for searching genome databases that shortlists potential matches for a given query sequence by evaluating it against a pre-generated index of the sequence database. The main contributions of our work are as following;

1. We incorporate the information from data sequences and past queries for index construction. Information extracted from data sequences facilitates our index to capture accurately the part of the database which represents the database most. Information from past queries allows the index to capture those subsequences that are representative in the past queries but not in the database.

2. We propose the Distance-based Sequence Indexing Method (DSIM), where data subsequences are represented as edit distances with respect to some pre-selected reference words appearing in the sequence. This scheme saves a significant amount of string matching cost during query processing. We also propose incremental index update algorithms to efficiently update the index when new data are inserted and new queries come along.

3. We have implemented the algorithms as a standalone program and as well as a part of BLAST, and conducted extensive experiments on a 2.62GB genome database. The results show that our algorithm outperforms BLAST by over 200 times in speed for long queries and achieves the same performance for short queries, without sacrificing the quality of the results.

The remainder of this paper is organized as follows. The next section gives a background on genome searching and related work. Our framework and associated algorithm are introduced in Section 3. Section 4 gives the cost model to study the effectiveness and efficiency of DSIM. Following that, experimental results are presented in Section 5, and Section 6 concludes the paper.

2 Background

Most of the operations on genomic databases involve searching for database sequences that are similar to a given query sequence. This section first examines BLAST – the

Proceedings of the 5th IEEE Symposium on Bioinformatics and Bioengineering (BIBE'05)
0-7695-2476-1/05 $20.00 © 2005 IEEE

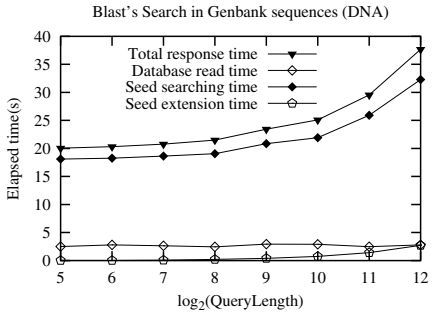


Figure 1. Breakdown of BLAST's search time

standard tool that is being used by biologists – before discussing other related work.

2.1 BLAST – The Standard Tool

BLAST [4, 5] is the standard tool that molecular biologists use to search for sequence similarity in genomic databases, and is known to be one of the more efficient tools available.

The operation of matching a query sequence against a library of data sequences is carried out in three steps. First, BLAST generates a list of query sliding windows of length k (k-tuples), where k is a tunable parameter. In the second step, BLAST scans through every database sequence to see if it contains a k-tuple that can match with one of the query k-tuples to produce a *seed* with a score greater than or equal to a predetermined threshold T. Finally, the seeds are extended on the both sides to locate longer similar segment pairs. BLAST permits a tradeoff between speed and sensitivity, by adjusting the settings for the threshold T and the length of k-tuple. For DNA queries, the default settings are $T = 0$ and $k = 11$.

To understand the cost breakdown, we recompiled the source codes of BLAST (downloaded from [1]) with the profiling option and ran the program with a profiler, gprof [3] to generate statistics on the time spent on each function. These statistics are grouped to produce the time spent on database retrieval, k-tuple matching, and seed extension.

We ran BLAST on a DNA database of 487 KB sequences with 1.7 GB letters to study the effect of query length on BLAST. For each query length, we generated 100 random queries and obtained the average running time. The results, plotted in Figure 1, show that the seed searching time dominates the total response time for BLAST. Database read time and seed extension time are just a small portion of the total response time.

2.2 Related Work

As described earlier, BLAST is an exhaustive search tool that checks all the entries in the database to formulate an answer set to a query; other similar tools include SSearch [15] and FastA [14]. The alternative to exhaustive search is to use index-based approaches [9, 6, 10, 16, 7, 8, 13]. We review several such schemes below.

RAMdb [9] is a system for finding short patterns called *motifs* in genome databases. It requires a large index that is twice the size of the original flat-file database including the textual descriptions and suffers from a lack of special-purpose ranking schemes for identifying initial match regions. In addition, the non-overlapping interval of query motifs can lead to false dismissals.

Notation	Meaning
n	total size of data sequences
m	size of a query sequence
k	substring length
$RefWordSet$	reference word set
ϵ	edit distance threshold of a seed match
$edit\_dist$	edit distance function on two strings

Table 1. Algorithm parameters

The FLASH search tool [6] redundantly indexes genome data based on a probabilistic scheme . FLASH was fast, but the hash-table index used is uncompressed and impractically large – For a nucleotide collection of around 100 MB, the index requires 18 Gb on disk, around 180 times the collection size.

SST [10] uses the tree structured vector quantization technique for sequence searching. SST is only effective for finding data sequences that are very similar to a query sequence.

In [11], DNA sequences are transformed into numerical vector spaces to allow the use of multi-dimensional indexing approaches for sequence similarity search. Though the method avoids false dismissals and offers very fast filtering, their drawback is that the approximation of edit distance is not sufficiently tight, which increases the cost of refining results for final output.

CAFE [16] is based on a partitioned 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 subset of database sequences with the query. CAFE bears the additional overhead of uncompressing the index at runtime for a query, which we do not have. Although it is computationally efficient than the exhaustive methods, "exhaustive systems generally have better retrieval effectiveness".

Compared to our previous work [7, 8] in DNA sequence indexing and searching, DSIM is much more efficient with the slight cost of some searching accuracy.

3 The Proposed Method

In this section, we describe our algorithm design, including the proposed framework, index construction, incremental index update and query processing. The notations, which will be explained as they are used, are summarized in Table 1.

3.1 Basic Framework of DSIM

The basic idea of DSIM is to compress a data sequence by representing the consecutive overlapping sequence substrings with their respective edit distances to some instance in a reference word set. The task of searching similar data sequences to a query sequence is decomposed into two steps: First, we examine the compressed data sequences to find the matches between the query substrings and reference words;

the pre-computed edit distance permits this step to be performed quickly. Next, sequences with significant matches are post-processed for actual sequence match and alignment using some approaches such as BLAST.

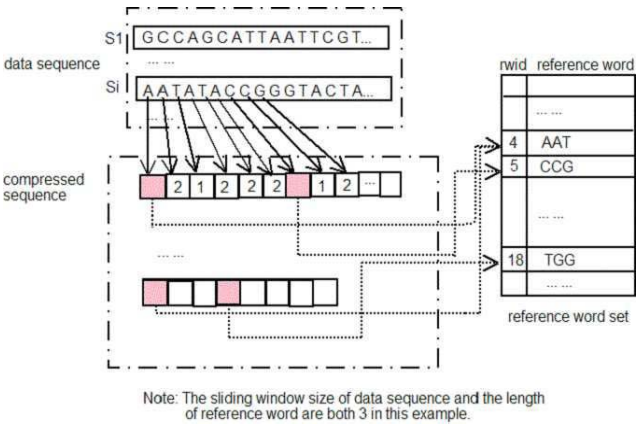


Figure 2. A structure of DSIM

We borrow the idea from the MPEG-based video compression [2] which divides video data into three types of frames: I-frame(Intra Frame), P-frame(Predicted Frame) and B-frame(Bidirectional Frame). An I-frame contains the full image representation; a P-frame is the difference to the previous I-frame; B-frames are coded as differences from the last or next I-frame or P-frame. This compression technique significantly reduces data storage size, and speeds up processing time while maintaining satisfactory accuracy. In genome sequence database, we select some reference genome subsequences, called reference words which are similar to I-frames in video sequence and for the genome subsequences following the reference word, only the edit distance from the reference word is stored. Figure 2 illustrates an example of the DSIM transformation and search structure. In this figure, we describe how the data sequence S_i is transformed to the compressed sequence based on the reference word set when the size of the sliding window is set to 3. Since the first sliding window of data sequence S_i, AAT is a reference word(rwid=4), the following sliding windows ATA, TAT, ATA, TAC and ACC are denoted as the edit distance from the reference word AAT until next reference word CCG(rwid=5) occurs. The same principle can be applied to the remaining sliding windows of the sequence S_i and other data sequences.

In our problem domain, we aim to reduce the substring matching time by sequence compression. Brute force substring matching algorithms build a lookup table on the query substrings and find matches for each data substring, resulting in an average cost of $O(mn)$. BLAST employs an optimized algorithm using bit encoding and parallel lookup but it still grows sublinearly to $O(mn)$. In contrast, at the price of a little memory overhead, we build a search structure on the reference word set so that we only need to look up a fraction of substrings within the data sequence where the target reference word occurs. As shown in Section 4, the resulting processing time grows only sublinearly to $O(n)$.

The effectiveness of DSIM relies on the proper choice of the reference word set. We propose two heuristics-based strategies. The first strategy is to choose the frequently-occurring data substrings within the data sequences. The heuristic is that a long significant segment match between two sequences is likely to consist of multiple consecutive short substring matches. Thus a higher number of matches with the frequently-occurring data substrings indicates a greater similarity. This heuristic favors queries that are similar to the high-frequency data subsequences. However, it may fail for queries that contain few or no frequently-occurring data substrings. Our second heuristic is based on the observation that some parts of past queries are likely to re-appear in future queries. Thus we add substrings in the past queries to the reference word set. This heuristic is different from the strategy of caching search results for the future use, which needs to compare future queries with past queries for local similarity match and again introduces overheads. We employ a combination of the two heuristics to maintain a reference word set that is updated dynamically for new incoming data and queries. Note that our framework can also accommodate other heuristics for determining the reference word set. Figure 3 illustrates the high-level framework described above.

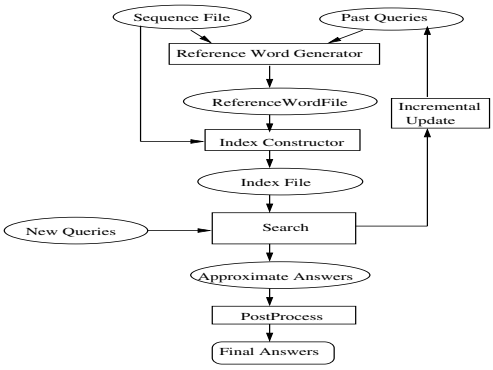


Figure 3. Flow chart of DSIM

3.2 Index Construction

DSIM consists of two components: a reference word set and a compressed data file. The initial index construction encodes the data sequences, extracts the high-frequency data substrings, and creates the reference word set and compressed sequences. In Section 3.1, we have discussed how to construct the reference word set. In this section, We will mainly discuss how to generate the compressed data file by using the following two steps, sequence encoding and sequence compression.

3.2.1 Sequence Encoding

Genome data sequences are usually stored as strings in FASTA format. Encoding data sequences into integers allows us to perform substring matching efficiently and to speed up incremental index updates. To encode k-length

substring, the transformation function $str2Int$ first translates the substring into a bit sequence by mapping every character into a unique bit pattern. The bit sequence is then converted into a non-negative integer using a bit-OR operation with an integer mask of $2^{2k} - 1$. We assume that the DNA alphabet is {A, C, G, T} and substrings containing 'N' letters which represents unknown values are encoded with an indicator -127. We assign the bit pattern as follows: 00 for 'A', 01 for 'C', 10 for 'G', and 11 for 'T'. For example, given a DNA substring "ACGTT", the bit sequence is 0001101111 and the integer mask is 1023, and thus $str2Int$("ACGTT") = 111.

We also define the function ed to be the edit distance [12] between two strings represented in their encoded values. That is, for two strings s_1 and s_2, $edit\_dist(s_1, s_2)$=$ed(str2Int(s1), str2Int(s_2))$.

The transformation function $str2Int$ is lossless for strings of length up to 15 in a 32-bit system architecture since one bit is used as sign bit. This length limit does not constrain the usability of our algorithm, as the default substring length used in BLAST is 11 and consecutive substring matches longer than 15 can be handled through multiple overlapping substring matches of length 15 or shorter.

3.2.2 Sequence Compression

Sequence compression is performed with the reference word set and the encoded data sequences. Assuming that the set of past queries is empty initially, the initial reference word set is the set of frequently-occurring substrings in the data sequences. We employed a simple double-scanning-based algorithm to find those data substrings.

Algorithm 1 Compress
Input: $RefWordSet$, $EncodedFile$
Output: $CompressedFile$
Method:

```
1:  Create the search structure RefWord_B+ on RefWordSet;
2:  for each sequence S with (SeqId,seqLen) do
3:     output(CompressedFile, seqId);
4:     output(CompressedFile, seqLen);
5:     for each v_i at position i of sequence S in EncodedFile do
6:        if v_i==-127 then
7:           /*indicating the substring contains 'N' letters*/
8:           output(CompressedFile,-127);
9:        else
10:          if v_i==lastRef or Search(RefWord_B+, v_i) then
11:             output(CompressedFile,v_i);
12:             lastRef = v_i;
13:          else
14:             output(CompressedFile,-ed(v_i,lastRef));
15:          end if
16:       end if
17:    end for
18: end for
```

Prior to sequence compression, we first read in the reference words, encode them and then build an in-memory B^+-tree, $RefWord\_B^+$ on the encoded values. If the size of the reference word set is too large for the B^+-tree to fit entirely in memory, we split the reference words into batches and process them in turn: For the first batch, we run Algorithm 1 to create the compressed sequences. From the second batch onwards, we update the compressed sequences by executing the index update algorithm.

As we scan the encoded data sequences, we check whether the encoded value v at each position occurs in the reference word set by the function $Search(RefWord\_B^+, v)$. If so, the encoded value is replaced by the identity of the matched reference word; otherwise the function ed is used to compute the edit distance between the encoded value and the previous reference word, and the distance replaces the encoded value as the compressed value. If no reference word has been found previously, the nearest next instance of some reference word is used. To distinguish between the two types of compressed values, we store edit distances as negative values since substring encoding only returns non-negative values.

Compressed values are sequentially stored in the compressed sequence file. Each sequence in the compressed file starts with a two-field header containing the sequence's starting ID, seqId, and length, seqLen, in the original data file. Due to the one-to-one correspondence, the compressed file has the advantage that matching query patterns identified in a compressed sequence implicitly tell the position of the matches in the original data sequence. Note that this file organization is the same for the encoded sequence file. Figure 4 gives the corresponding compressed format for the DNA sequence.

Data sequence	…ATTGAAACGGGCAATATGTC…
Reference words	{ATTGAAACGGG, AACGGGCAATA, GGGGCAATATG}, encoded as 1015914, 108812, 2789582
1. Sequence substring extraction	**ATTGAAACGGG**, TTGAAACGGGC, TGAAACGGGCA, GAAACGGGCAA, AAACGGGCAAT, **AACGGGCAATA**, ACGGGCAATAT, CGGGCAATATG, GGGCAATATGT, GGCAATATGTC
2. Substring encoding	**1015914**, 4063657, 3671716, 2103952, 27203, **108812**, 435251, 1741006, 2769723, 2690285
3. Substring compression	**1015914**, *ed*(**1015914**, 4063657), *ed*(**1015914**, 3671716), *ed*(**1015914**, 2103952), *ed*(**1015914**, 27203), **108812**, ed(**108812**, 435251), *ed*(**108812**, 1741006), ed(**108812**, 2769723), *ed*(**108812**, 2690285)
4. Distance index creation	**1015914**, -2, -4, -6, -8, **108812**, -2, -4, -6, -8

Note: The underlined and bolded substrings or numbers are occurrences of Reference words. *ed* is the edit distance function on two integers.

Figure 4. Index building example

The algorithm for initial sequence compression is given in Algorithm 1. For efficiency, the edit distances are always computed from the previous occurrence of a reference word. Comparing with next occurrence of a reference word can be easily done by buffering current values until the next reference word is encountered, and then compute the distance for every buffered value.

In addition, based on the property of edit distance saying that for two strings s_1 and s_2, $edit\_dist(s_1, s_2) \leq max(length(s_1), length(s_2))$, the maximum word distance is 15 since an encoded substring is restricted to be no longer than 15 in our algorithm. Therefore we use one byte to store

the distance value, instead of one integer, which reduces the index size.

3.3 Incremental Index Update

As the sequence database is updated and new queries come along, the reference word set and the compressed file need to be updated accordingly. There are four cases of index updates:

Case I: Collection of new queries

For each query newly inserted into the past query collection, we search for every unique query substring in $RefWord\_B^+$. If the query substring is not found, we add it to the reference word set and its $RefWord\_B^+$.

The compressed file has to be updated according to the new reference words. The update affects only the substrings that are not in reference word set. The encoded values of those substrings are retrieved from the sequence encoding file, and matched against the new reference words. If there is a match, then the compressed value for the corresponding substring is changed from an edit distance to an encoded value. The algorithm for updating the reference words set and the compressed file according to the new queries collection are similar to the one for building them, we will not depict them for brevity.

Case II: Insertion of new sequences

After inserting the new query sequences, we first append the new encoded data sequences to the encoded sequence file. Next we extract the frequently-occurring substrings, i.e. new reference words from each sequence. Finally, the compressed sequence file and the reference word set are updated based on the new reference words in the similar way of Case I.

Case III: Modification of existing sequences

First, the old encoding within the encoded sequence is overwritten with the encoding of the modified sequences. Following that, we treat the modified data sequences as new data sequences and apply the same steps in Case II.

Case IV: Deletion of existing sequences

We delete the deleted sequences from the encoded file. If the deleted sequences include the reference words, the reference word set and compressed file will be updated accordingly. Note that after an incremental update, there may exist some "dangling" reference words that no longer appear in the database sequences. These dangling references can be removed by periodically scanning the database to count their occurrences. This process does not affect the compressed sequence file.

3.4 Query Processing

In this section, we will describe how DNA homology search is performed using the proposed index structure, DSIM. Given a query sequence, the unique query substrings, QWs, are extracted and attached to the similar reference words within edit distance ϵ using the function $AttachQW(RefWord\_B^+, QW, \epsilon)$. For each data value v_i, in the compressed file, if it is an occurrence of some reference word, the attached list of similar query substrings $qwlist$ will be returned using the function $SearchRefWord(RefWord\_B^+, v_i, qwlist)$, where each query substring is called a "seed match" to current data value. After finishing scanning each data value in compressed file, we compute the overall score for the match pair of data value and query sequence in $MatchPair$ using scoring function. The query results are those sequences with top K scores. Post-processing is a desirable or necessary step to align the search results to the query sequence using an algorithm such as BLAST. Note that if no query substring can be attached to the $RefWord\_B^+$, we directly run normal BLAST search to get the final alignment results. The detailed search algorithm is given in Algorithm 2.

Algorithm 2 SeqSearch
Input: $querySeq$, ϵ, $RefWordSet$, $CompressedFile$
Output: $topK$
Method:

```
1:  Create search structure RefWord_B+ on RefWordSet;
2:  Extract unique query substrings, QWs with their positions from querySeq;
3:  for each unique query substring QW_i ∈ QWs do
4:      AttachQW(RefWord_B+, QW_i, ε);
5:  end for
6:  if none of QWs is attached to RefWord_B+ then
7:      run normal BLAST and exit the algorithm.
8:  end if
9:  for each sequence S with (seqId, seqLen) do
10:     MatchPair = null;
11:     for each value v_i at position i of sequence S in CompressedFile do
12:         if v_i ≥ 0 then
13:             curRefWord = v_i;
14:             if SearchRefWord(RefWord_B+, v_i, qwlist) and
                qwlist! = null then
15:                 MatchPair+ = (qwlist, seqId, i);
16:                 lastRefWord = v_i;
17:                 lastqwlist = qwlist;
18:             end if
19:         else if −v_i ≤ ε && curRefWord == lastRefWord then
20:             MatchPair+ = (lastqwlist, seqId, i);
21:         end if
22:     end for
23:     Compute the score for sequence S with (seqId, seqLen) and query substrings
        in MatchPair using scoring function;
24: end for
25: Return topK, top K sequences aligned with querySeq.
```

"Seed match" without errors: "Seed match" includes two kinds of exact matches between query substrings and reference words, and between the data substrings and reference words. This scheme is fast in searching in reference word set as it can be implemented as searching in a B^+-tree, $RefWord\_B^+$. We can also ignore all compressed data substrings that are not reference words. The disadvantage of this scheme is the miss of substring matches that are not reference words. In practice, our experiments show that this method is effective for queries whose substrings are well captured in the reference word set.

"Seed match" with errors: If a query substring and a data substring match respectively to the same reference word within edit distance ϵ, we consider this as a "seed match" with errors. This guarantees no miss of substring matches that are different from reference words with at most edit distance ϵ. Thanks to the pre-computed edit distances in the compressed sequences for data substrings not belonging to reference word set, we simply compare them with ϵ to identify matches. Note that we are interested in the positions of a seed match rather than the contents of the match. Therefore, once an ϵ distance match is found at position i in a com-

Notation	Meaning
t	# of bytes in an integer. Default to 4.
$C_{I/O}$	Avg. I/O time for fetching a data element
$C_{refword}$	Avg. search time in reference words
C_{edit}	time for edit distance computation for k-long strings
m_{uniq}	# of unique query substrings of length k
p	% of data substrings that are reference words
S_{index}	Size of our index
T_{index}	Index build time
R	Response time of our search algorithm

Table 2. Cost parameters

pressed sequence, we do not have to find in the encoded sequence to know the data substring at position i. We combine the use of both approaches in our DNA homology search method. If ϵ is set 0, the first approach is used otherwise the second.

Scoring functions: Scoring sequences based on the seed matches allows us to filter out data sequences with low similarity and perform post-processing only on the highly similar sequences. CAFE [16] has proposed a few useful scoring functions and shown their effectiveness. For a set of seed matches with the same relative position offset between a data sequence and query sequence, CAFE's various scoring functions consider these different factors: the number of matches, the number of non-overlapping matches and the span of matches in the data sequence. In our implementation, the number of matches is used as the scoring function. Nevertheless, our algorithm can also incorporate other scoring functions mentioned above because we capture the positions of the seed matches.

4 Cost Analysis

To study the effectiveness and efficiency of DSIM, we present a cost model in this section. The main concern is the search time, while index size and index build time tend to be secondary due to the nature of applications. The cost parameters are shown in Table 2.

Index size: The compression file consists of two types of elements: the occurrences of reference words, stored as an integer, and distances of data substrings not belonging to the reference word set, stored as a byte. Therefore, the size of compression sequence is,

$$S_{index} = (t \times n \times p) + (1-p) \times n = (1+3p) \times n \quad (1)$$

Index build time: The index build time is composed of the I/O time in reading in sequences and outputting sequence compression, and the lookup time taken for searching every data substring in the B^+-tree of reference words. Hence, the index building time is:

$$T_{index} = n \times C_{I/O} + (n-k+1) \times C_{refword} + (1+3p) \times n \times C_{I/O} \quad (2)$$

Search time: DSIM's query processing consists of two steps: (1) matching and attaching query substrings to reference words, and (2) scanning the sequence compression to find occurrences of reference words. For step (1), if ϵ is set to 0, there is a single lookup in the B^+-tree built on $RefWordSet$ for each unique query substring. The cost is $m_{uniq} * C_{refword}$. To study the worst case scenario, assume there is only a single cluster of reference words. Then we have to compute the edit distance for every pair of query substrings and reference word. The cost taken is therefore $m_{uniq} * |RefWordSet| * C_{edit}$.

$$R_{attach} = \begin{cases} m_{uniq} \times C_{refword} & \text{if } \epsilon = 0 \\ m_{uniq} \times |RefWordSet| \times C_{edit} & \text{if } \epsilon > 0 \end{cases} \quad (3)$$

Scanning of sequence compression and look up, incurs both the I/O cost and B^+-tree lookup cost. There are exactly $|RefWordSet|$ number of lookups for the exhaustive scanning of all the data. The cost taken for step (2) is, therefore,

$$R_{scan} = (1+3p) \times n \times C_{I/O} + |RefWordSet| \times C_{refword} \quad (4)$$

and the response time of our search algorithm is

$$R = R_{attach} + R_{scan} \quad (5)$$

From Equations (3), (4) and (5), it can be derived that the search cost of DSIM is bounded by $O(m \times |RefWordSet| + (1+3p) \times n)$.

5 Performance Study

Having introduced our sequence matching algorithms, we will present the experimental results to profile its performance characteristics. The experiment environment is described in Section 5.1. The genome sequence search efficiency and search accuracy of our proposed method are studied in Section 5.2 and Section 5.3, respectively. For comparison purposes, we also include the BLAST tool in our performance study.

5.1 Experiment Setup

The experiments were implemented on a Sunfire 4800 machine with 12 Ultrasparc3 CPU of 750MHz, 16GB free memory and 70GB free hard disk. The data are human genome sequences downloaded from NCBI. We used 6 data sets described in Table 3 to conduct various experiments. Unless explicitly stated, the dataset number 6, the largest in size (2.62GB), is always used for the following experiments. The query sequences are segments randomly extracted from the genome sequences, and the reported performance results are the average over 10 queries.

Data Set No.	Data collections	No. of sequences	Total Size
1	Genome 2	130	230MB
2	Genome 1-2	267	444MB
3	Genome 1-4	670	812MB
4	Genome 1-10	1196	1684MB
5	Genome 1-15	1530	2136MB
6	Genome 1-22,X,Y	2007	2621MB

Table 3. Description of data set

A list of experimental parameters, together with their default values, are summarized in Table 4.

5.2 Search Efficiency

In this section, we will study the genome sequence homology search efficiency of our method when varying the

Notation	Meaning	Range	Default
k	substring length	7-15	11
$qLen$	query length	32-16384	1024
qNo	number of queries	10-50	10
ϵ	seed match error threshold	0-5	0
$refWordNo$	no. of reference words	200-16000	8000

Table 4. Experimental parameters

parameters, such as the query length and edit distance threshold for seed matches. Moreover, the comparison of search efficiency between BLAST and our method is also studied.

Varying query length: we set k=11, qNo=10, $refWordNo$=8000, and ϵ=0. The average response time of running 10 queries is shown in Figure 5(a). It confirms the earlier cost analysis that our search algorithm is almost invariant to the query length because the query substrings are directly compared with reference words for similarity match. Unlike BLAST, the data sequences in database are not compared to query substrings. Instead, only those data substrings which are the reference words, are searched in the reference word set to get the attached similar query substring lists. Hence, our algorithm obviously outperforms BLAST much more with the query length increasing.

Varying edit distance threshold for seed matches: We set k=11, $qLen$=1024, qNo=10 and $refWordNo$=8000. Since BLAST does not directly support seed match with errors, we therefore use the "effective seed length" for BLAST search, where effective seed length=$k - \epsilon$. This is reasonable since the set of exact seed matches of length $k - \epsilon$ is a subset of the set of seed matches of length k with error threshold ϵ. Figure 5(b) shows the search performance of our method and BLAST with respect to varying edit distance threshold for seed matches. We observe that DSIM outperforms BLAST by up to a speedup factor 4 and also remains consistent search time with respect to the edit distance threshold because increasing the threshold only affects the first step of DSIM search and causes more query substrings to be attached to reference words. But the second step, scanning of compressed files, is not affected much because edit distance threshold does not cause more lookup of data substrings in the reference words. Actually, the number of lookups is only controlled by the number of occurrences of reference words in the query sequence.

Varying database size: We set k=11, $qLen$=64, qNo=10, $refWordNo$=8000 and ϵ=0. Figure 5(c) shows the effect of data size on the search performance of both methods. The result shows that the search time of DSIM grows moderately super-linear to the database size. The reason is that the amount of reference word occurrences in the database grows faster with the increasing database size since the reference word set contains the frequent-occurring data substrings. Consequently, this incurs fast growth of the cost of lookups in the reference word set during DSIM's search. As shown in Figure 5(c), the increase in query time for BLAST is much more significant than DSIM method as BLAST suffers from higher I/O cost in large sequence databases. It again confirms the cost analysis.

5.3 Search Accuracy

Many existing methods have achieved good query performance by relaxing the requirement of accuracy. However, due to the nature of genomic applications, compromise on the accuracy could be a big risk to take. Since BLAST is well studied and widely used by biologists, we will evaluate the search accuracy of our method by comparing with BLAST. The search accuracy metrics, $Precision$ and $Recall$ are studied for our methods. And we also give the analysis about how the edit distance between the queries and reference words affects the accuracy of DSIM.

Precision: In our experiments, we evaluate the search precision of DSIM by checking how big the intersection of the hits returned by BLAST and DSIM is when the top K hits of both methods are considered. The ratio of the intersection hits of BLAST and DSIM ($Intersection\_Ratio$) in top K hits will be used as precision to measure the genome homology search accuracy of DSIM. We also evaluate how $Intersection\_Ratio$ changes by varying the edit distance.

$$Intersection\_Ratio = \frac{number\ of\ hits\ in\ intersection}{K}$$

We set k = 11, $qLen$ = 64, $refWordNo$ = 8000, qNo = 10 and ϵ = 0. Figure 6(a) plots the average $Intersection\_Ratio$ between BLAST and DSIM with the different "edit distance" for query sequences with length 64, in which "edit distance" means the edit distance between query and reference word. With "edit distance" equal to 1, 2 or 3, $Intersection\_Ratio$ is above 60% when K is greater than 30. The experiment result also shows that $Intersection\_Ratio$ will decrease when increasing the edit distance between query sequence and reference word. It means that the portion of common hits between BLAST and DSIM will decrease when the query sequence is very different from the reference words set.

Recall: The recall of DSIM is also studied in our experiment. We set k=11, $qLen$=64, qNo=10, ϵ=1 and $refWordNo$=8000.

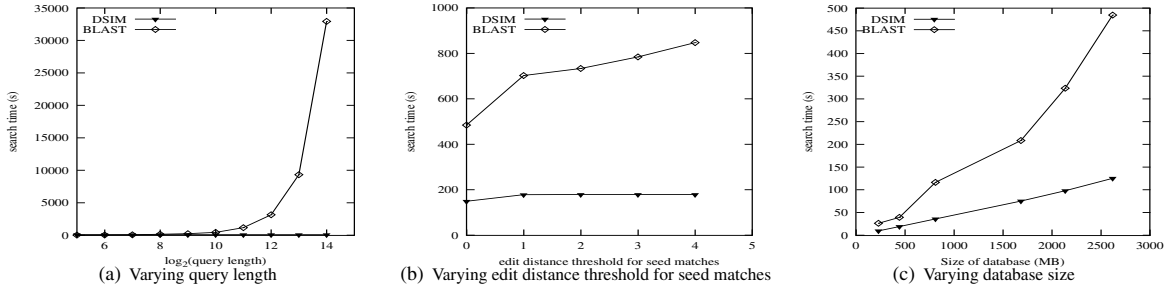
$$Recall = \frac{Number\ of\ matching\ sequences\ returned}{Total\ number\ of\ matching\ sequences}$$

Figure 6(b) shows the recall of selecting the top 20 search results of BLAST, for the various number of hits retrieved by our algorithm. It can be seen that DSIM quickly achieves a recall of 90% in the output of 60 hits, and is able to return the hits as accurate as those of BLAST eventually.

Varying edit distance of the query to reference word set: As we discussed earlier, the search performance of DSIM is dependent on how close the query is to the reference words. If none of query substrings matches with any reference word, DSIM does not process the query, and instead run the normal BLAST search. In this experiment, we studied how different a query can be from the reference word set, while keeping DSIM effective. The distance of a query to a reference word set, is measured as the minimum of the query's edit distance to the reference word set. We choose k=11, $qLen$=64, qNo=10, ϵ=1 and $refWordNo$=8000. The number of answers returned by DSIM is set as 60. Figure 6(c) shows that DSIM achieves a recall of 60% for the edit distance of 15. In other words, a query can be up to 23% (derived by $\frac{15}{64}$) different from the reference word set, to keep DSIM being effective at a recall of 60%. To make DSIM effective, the index file is required to be updated after adding the past queries into the reference word set when the recall falls below the acceptable level.

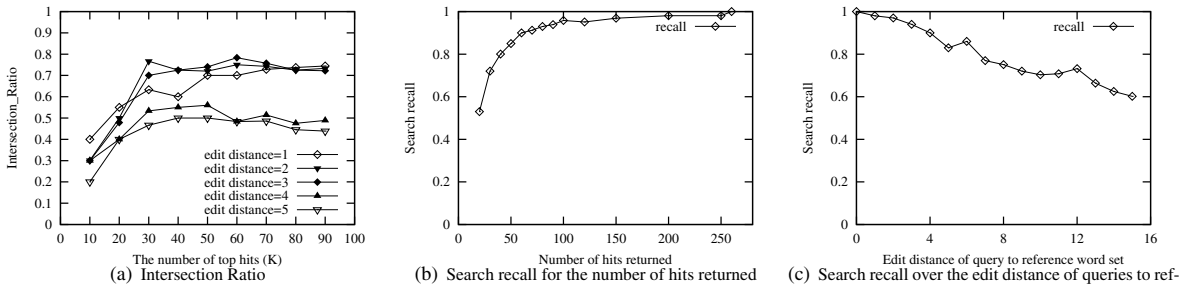
6 Conclusion

In this paper, we have presented a novel method called DSIM for identifying sequence homology in large genomic



(a) Varying query length (b) Varying edit distance threshold for seed matches (c) Varying database size

Figure 5. Experimental Results for Query Efficiency



(a) Intersection Ratio (b) Search recall for the number of hits returned (c) Search recall over the edit distance of queries to reference word set

Figure 6. Experimental Results for Query Accuracy

databases. The method is designed to speed up query processing, by maintaining in memory only a selected set of high-frequency data segments, and by reducing the remaining data segments to "deltas" from the selected segments in such a way that all data sequences that match a given query sequence can be shortlisted from this compact index. Experimental results on real data sets confirmed that the proposed algorithm outperforms existing tools such as the latest version of BLAST, significantly lowering processing time without compromising the quality of the results.

Acknowledgement: We thank Hao Wang for his contribution in implementation and performance evaluation, and our colleagues in Genome Institute of Singapore for useful discussion.

References

[1] ftp://ncbi.nlm.nih.gov/blast/db/readme.

[2] http://mpeg.telecomitalialab.com/.

[3] http://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[4] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. A basic local alignment search tool. *Journal of Moelcular Biology*, 215:403–410, 1990.

[5] S. F. Altschul, T. L. Madden, A. A. Schaeffer, J.Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psiblast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25:3389–3402, 1997.

[6] A. Califano and I. Rigoutsos. Flash: A fast look-up algorithm for string homology. In *Proceedinsg of the International Conference on Intelligent Systems for Molecular Biology*, pages 56–64, Bethesda, MD, 1993.

[7] X. Cao, S.C. Li, B.C. Ooi, and A.K.H. Tung. Piers: An efficient model for similarity search in dna sequence databases. *ACM Sigmod Record*, 33, 2004.

[8] X. Cao, S.C. Li, and A.K.H. Tung. Indexing dna sequences using q-gram. In *Proc. 2005 Int. Conf. Database Systems for advanced applications*, pages 4–16, Beijing, China, Apr. 2005.

[9] C. Fondrat and P. Dessen. A rapid access motif database (ramdb) with a search algorithm for the retrieval patterns in nucleic acids or protein databanks. *Computer Applications in the Biosciences*, 11(3):273–279, 1995.

[10] E. Giladi, M. C. Walker, J. Z. Wang, and W. Volkmuth. Sst: An algorithm for searching sequence databases in time proportional to the logarithm of the database size. In *Proceedings of RECOMB'00*, 2000.

[11] T. Kahveci and A. Singh. An efficient index structure for string databases. In *Int. Conf. VLDB*, Roma, Italy, 2001.

[12] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR*, 163(4):845–848.

[13] C. Meek, J.M. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences. In *Proc. 2003 Int. Conf. Very Large Data Bases (VLDB'03)*, pages 910–921, Berlin, Germany, Sept. 2003.

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

[15] D. J. States, W. Gish, and S. F. Altschul. Improved sensivity of nucleic acid databas searches using application-specific scoring matrices. *Methods: A Companion to Methods in Enzymology*, 3(1):66–70, 1991.

[16] H. Williams and J. Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 2002.