Piers: An Efficient Model for Similarity Search in DNA Sequence Databases

Xia Cao Shuai Cheng Li Beng Chin Ooi Anthony K.H. Tung

Department of Computer Science, National University of Singapore, Singapore, 117543
Email: {caoxia,lisc,ooibc,atung}@comp.nus.edu.sg

ABSTRACT

Growing interest in genomic research has resulted in the creation of huge biological sequence databases. In this paper, we present a hash-based pier model for efficient homology search in large DNA sequence databases. In our model, only certain segments in the databases called 'piers' need to be accessed during searches as opposite to other approaches which require a full scan on the biological sequence database. To further improve the search efficiency, the piers are stored in a specially designed hash table which helps to avoid expensive alignment operation. The hash table is small enough to reside in main memory, hence avoiding I/O in the search steps. We show theoretically and empirically that the proposed approach can efficiently detect biological sequences that are similar to a query sequence with very high sensitivity.

1. INTRODUCTION

DNA is the basic blueprint of life, and its structure can be viewed as a simple but very long sequence over the four-letter alphabet of 'A', 'C', 'G' and 'T'. Homology search in DNA databases is an important function as it is useful for discovering the location of functional sites, searching the existence of novel repeats and conducting the comparative analysis of different DNA sequences. To cater for the evolutionary mutations of genomic sequences and noise in the data, approximate sequence matching is preferred to exact matching in genomic databases.

Many algorithms have been developed for this task, with the most fundamental one being the Smith-Waterman alignment algorithm [14], which is a dynamic programming approach for finding an optimal alignment between a query and its target sequence. Such an approach takes $\Theta(mn)$ time, m and n being the length of the two sequences respectively. This is obviously too slow for searching large, external memory sequence databases. In view of this, various alternative approaches have been proposed.

A most common approach to improve efficiency revolves around the idea of filtering and refinement [13, 1]. In such approaches, the sequences database is first broken into short segments and matched against the query sequences. Segments with low similarity are first filtered off while more complex computations are done on the remaining high similarity segments to form the final result. The efficiency and sensitivity of such approaches are highly dependant on the choice for the length of the segments and how regular the segments are being sampled from the database sequences. For example, in the case of BLAST [1], a segment length of 11 characters is usually used and segments are obtained from every position in the database sequence.

In this paper, we propose to use the hash based pier model for efficient and sensitive sequence search. Our technique focuses on effective filtering in the first phase of the search since the performance in the second phase is typically similar across most of such approaches. The pier p in our model is defined as a segment with length ℓ_p and located at position pos in a data sequence. [1]

During pre-processing, the piers are randomly picked from a data sequence S based on the principle that at least one pier is contained within any subsequence of S having a minimum length. These piers are then stored in a hash table for efficient access. Such an approach gives us a much lower pre-processing time compared to BLAST and index building approaches.

During query time, by picking each seed generated from the query sequences and enumerating its neighbors (i.e. segments of the same length that are with a small edit distance from it), candidate buckets can be located in the hash table very efficiently. With the algorithm, we can enumerate all the neighbors which are potential candidates without searching through the whole hash space. All subsequences which contain at least one matched pier will be be compared against the relevant portion of the query sequence to verify their similarity. Both the efficiency and sensitivity of our search method will be studied in the paper.

The rest of the paper is organized as follows. We briefly review related work in Section 2. In Section 3, we will provide some definitions and formally give a problem statement. In Section 4, the pier model is proposed for DNA sequence search and the sensitivity of the proposed model is also analyzed. In Section 5, a hash-based pier model is presented for efficient sequence search. Section 6 shows how sequence similarity search can be efficiently processed using the proposed pier model with analysis on the space and time complexity of the method. Experiments are presented in Section 7. Section 8 concludes the paper.

2. BACKGROUND AND RELATED WORK

In this section, we review some work related to sequence similarity search for DNA sequence databases.

The most commonly known approach for DNA sequence search is that of a filter and refinement. In such approach, the database sequence is first scanned for short "seed" matches which are then extended into longer alignments. This method

[1]The name "pier" is selected since we believe that these small set of selected segments should be enough to "support" highly sensitive similarity search for the whose database sequence.

is used in programs like FASTA [13] and BLAST [1]. BLAST is a heuristic method for finding similar regions between two genomic sequences. It regards the exact match of w contiguous bases as candidates which are then extended along the left side and the right side to obtain the final alignments. But BLAST faces the dilemma of DNA homology search, which is that increasing seed size w decreases sensitivity whereas decreasing the seed size leads to too many random results. PatternHunter [8] is an improvement on BLAST both in speed and sensitivity. Essentially, PatternHunter's basic principles are similar to those of BLAST.

Other approaches use indexing structures like the suffix trees [9, 16]. The suffix tree and suffix array are popular data structures for exact sequence matching, as seen in algorithms like QUASAR [2]. Suffix trees, however, are good mostly for precise matches but are very awkward in handling mismatches [6, 11, 2]. Also they devour very large amounts of memory and disk usage. Oasis [10] is a novel search algorithm which uses a dynamic programming A*-search driven by a suffix tree index. Though it is faster, it also suffers the weakness of suffix tree.

There also exist some other index structures for biological sequence databases [4, 7, 17, 15, 12]. In [7, 12], some attempts have been made to transform DNA sequences into numerical vector space in conjunction with various multi-dimensional indexing approaches or other tree-structured index to do sequence similarity search. Though the false dismissal is avoided and the filtering processing is very fast, the drawback is that the approximation of the edit distance is not sufficiently tight, which increases the cost of refining the results to produce the final outcome. Williams et al. [17] proposed a search algorithm in a research prototype system, CAFE, which uses an inverted index to select a subset of sequences that display broad similarity to the query sequence. CAFE is faster, although less sensitive than BLAST when searching for very similar sequences. In [15], a new index for DNA sequences, called the *ed*-tree, is proposed to support probe-based homology search in DNA sequence databases.

A common problem among methods that use indexing structures is that they typically take up more pre-processing time to build the indexes. Furthermore, additional space is needed to store the indexes which can have the sizes ranging from 1 to 10 times the indexed sequences.

3. NOTATIONS AND PROBLEM STATEMENT

In this section, we will give some definitions and provide a formal problem statement.

3.1 Notations and Definitions

The most commonly used distance measurement for two sequences is referred as edit distance. It is a simple but fairly accurate measure for the evolutionary proximity of two DNA sequences [5]. The definition of edit distance is as follow:

DEFINITION 1. **Edit Distance**
The edit distance between two sequences is the minimum number of edit operations(i.e., insertions, deletions, and substitutions) of single characters needed to transform the first sequence into the second.

Notation	Description
$\|D\|$	the size of the DNA sequence database D
ℓ_p	the length of a pier
p_i	the ith pier sequence along D
ℓ_s	the length of a span
θ	edit distance threshold allowed for pier candidate
S	the data sequence in D
$\|S\|$	the length of sequence S
$S[i,j]$	the subsequence of S from i to j
$s \subset S$	s is the subsequence of sequence S
Q	the query sequence
$\|Q\|$	the length of query sequence Q
q_i	the ith query pattern in Q with length ℓ_p
ℓ_{min}	the minimum length of of the high similarity region
$edit(S,Q)$	the edit distance between two sequences S and Q

Table 1: The Parameters

In this paper, small segments from a sequence database are referred to as *piers*. Formally, a pier is defined as a tuple $\langle p, pos \rangle$, where pier sequence p is a segment of length ℓ_p extracted from a DNA sequence, and *pos* is the list of positions for the pier sequence p occurring in the data sequence.

For notation simplicity, we shall use p to refer to both the segment with the corresponding positions, and the segment itself. Based on the definition of piers, we define *span* to be the segment between two adjacent piers in the proposed pier model. Lastly, we define the set of target piers that we want to search to be *candidate*. The notations to be used in the paper are summarized in Table 1.

3.2 Problem Statement

The approximate sequence match problem can be classified into two categories: whole sequence matching and subsequence matching [3]. Since the subsequence matching problem is a generalization of the whole matching problem, we confine our attention to a subsequence match problem in this paper. The approximate subsequence matching problem can be described as follows:

PROBLEM 3.1. *Given sequence database $D = \{S_1, \ldots, S_d\}$ and query sequence Q, search all the subsequences in $S_i \in D$ so that the edit distance between data subsequence and the query subsequence is small or the alignment score is high. Normally, the score matrix for DNA sequence alignment is '+2' for match, '-1' for replacement and gap(or mismatch).*

We adopt the filter and refinement approach and due to the limit of space, will only focus on the filtering problem defined as follow:

PROBLEM 3.2. *Given edit distance threshold θ, find all candidates p, $p \subset S$ in data sequence S, $S \in D$ for each query pattern $q_i \subset Q$, where $edit(p, q_i) \leq \theta$.*

4. THE PROPOSED PIER MODEL

This section describes our pier model for biological sequence search. The main assumption in the pier model is that users are only interested in high similarity region that is of length greater than a minimum length l_{min}. Based on this assumption, we define piers in a biological sequence database as some segments in the sequence, which meet Property 4.1 defined below.

PROPERTY 4.1. **Pier Extraction Principle**
The Pier Extraction Principle *states that at least k piers should be contained in any subsequence with length no less than ℓ_{min}. This means that the following formula must hold: $((k+1)\ell_p + k\ell_s) \leq \ell_{min}$.*

Intuitively, the pier extraction principle simply ensures that consecutive piers are selected in the data sequence such that at least k of them will be contained in any subsequence of length no less than l_{min}. In the pier model, the pier sequences are extracted randomly from the data sequence as long as the pier extraction principle is satisfied. As explain later, this is done to minimize the probability of a high similarity region being lost in a worse case scenario.

By using piers, search results can be obtained efficiently and with acceptable sensitivity, without scanning the entire sequence database. The proposed pier model can be used as a general pre-processing model in sequence similarity search. This means that the piers are first extracted, then other existing similarity search methods are applied on the set of piers. We shall next analyze how our pier model is theoretically effective and sensitive enough for sequence search.

4.1 Sensitivity And Accuracy Analysis of the Pier Model

In approximate search, if a candidate is similar to the query sequence, then the edit distance of the corresponding parts in the alignment between query and candidate is small as well. Traditional methods are based on a more restrictive conclusion of this: if the distance between two sequences is short, then they have at least one length q segment that is exactly the same. In BLAST, the segment is referred to as seed, and in others, it may be referred to as q-gram, or pattern. Approaches based on this constraint cannot have seeds that are too long, otherwise the index structure will be large and it will also lead to low sensitivity. The seed length cannot be small as well, otherwise it makes filtration ineffective. In the case of BLAST, the seed length is typically set to 11. If the edit distance is allowed to be $10\% - 20\%$ of the matched subsequence, then it is possible that BLAST may miss some of the candidates. Gapped seed is used by PatternHunter [8] to reduce the missing candidates. The pier here is in some sense more flexible version of gapped seeds. Based on the observation that two similar sequences would have similar subsequences, we arrive at the following property:

PROPERTY 4.2. *If two sequences Q and C have edit(Q, C)$\leq \zeta$, then for each segment $s \subset C$, there exists a segment $s' \subset Q$ such that edit$(s, s') \leq \zeta$.*

By Property 4.2, we can simply index a partial set of segments of the database and when given each query segment s', search only for segments s in the index structure that has $edit(s, s') \leq \zeta$. Further, in above property, $edit(s, s')$ is rarely near to ζ when the segment length or pier length is much smaller than the length of S or Q. We may then state the property as follows:

PROPERTY 4.3. *If query Q and candidate subsequence C have edit(Q, C)$\leq \zeta$, then for each $s \subset C$, there exists a segment $s' \subset Q$ with high probability such that edit$(s, s') \leq \zeta'$ for some $\zeta' \leq \zeta$.*

For two subsequences with length $|S| = |Q| = L$ and $edit(Q, C) \leq \zeta$, if the edit distance between the pier randomly picked in S and the corresponding segment of Q of the alignment is ζ', the probability $P(\ell_p, L, \zeta', \zeta)$ for S to be found can be computed as following:

$$P(\ell_p, L, \zeta', \zeta) = \frac{\sum_{i=0}^{\zeta'} \binom{\ell_p}{i}\binom{L-\ell_p}{\zeta - i}}{\binom{L}{\zeta}} \qquad (1)$$

To illustrate the typical effectiveness of this argument for the experiment setting in this paper, we shall search for the candidate subsequence with length $L = 60$ and edit distance $\zeta = 6$ from the query subsequence. The length of pier, ℓ_p is set at 15, and $\zeta' = 3$. According to the above equation, a candidate will be found with the probability $P(15, 60, 3, 6) = 98.8\%$.

In Figure 1, the sensitivity with parameters $\zeta' = 3$, $\ell_p = 15$, $\zeta = 6$, which each query pattern covered at least two piers has been plotted. It is clearly that our proposed approach has high sensitivity. Note that in our case, we allow spans between piers. We can also slide the sequence to generate the piers. In a more intuitive manner, if two sequences C and Q have very few differences between them, then the probability of these differences being clustered in the same region should be low. Instead, the differences are expected to be scattered in most cases. By losing some of the rare cases, the computation cost can be reduced dramatically.

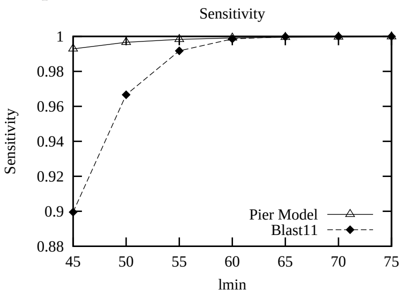


Figure 1: Sensitivity Analysis

5. THE HASH-BASED PIER MODEL

After the piers are extracted from data sequences, they are stored in a hash table $HTable$ to ensure efficient access. Given the query patterns, only buckets that have a high probability of holding similar segments are accessed. We called such buckets, *candidate buckets.*

In order to hash the piers to the pre-constructed hash table, it is necessary to encode the prefix of the pier with length λ, $\lambda \leq \ell_p$ into a 2λ bit integer. Each of the four possible nucleotides in a DNA sequence is encoded as two binary digits as follows:

$$f(a) = \begin{cases} 00 & a = \text{‘}A\text{’} \\ 01 & a = \text{‘}C\text{’} \\ 10 & a = \text{‘}G\text{’} \\ 11 & a = \text{‘}T\text{’} \end{cases}$$

Using the encoding function f, any λ-tuples of DNA sequence $s = b_1, b_2, ..., b_\lambda$ can be mapped uniquely to a 2λ bit integer by the encode function:

$$encode(s) = \sum_{i=1}^{\lambda} 4^{i-1} f(b_i)$$

The *encode* function is compact and efficient to process, but there is no way to encode any characters apart from the four valid letters. In our implementation, invalid characters are transformed into one of the four valid ones randomly in order to keep the positions of the matches found exactly as the positions in the original data sequence.

5.1 Construction of The Hash Table

After all buckets in the hash table are initialized as empty, each pier p_i will be inserted into the corresponding bucket in the hash table with the use of the hash function $encode(p_i[0, \lambda-1])$ which maps the first λ-tuple of p_i into a 2λ bit integer. The hash table has a total of 4^λ buckets for DNA sequences.

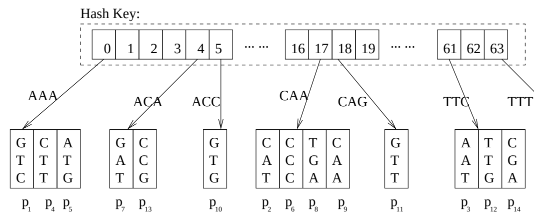


Figure 2: The Hash Table for Piers

The piers in the same bucket will share the hashed prefix, i.e., the same first λ-tuple. In average, there are $|P|/4^\lambda$ piers in each bucket for DNA sequences, where P is the set of piers.

EXAMPLE 5.1. *For example in Figure 2, $\ell_p = 6$, $\lambda = 3$, $P = \{p_1, p_2, \ldots, p_{15}\}$ and the hash table has $4^\lambda = 64$ buckets. In this example, all the 15 piers are inserted into the hash table one by one. The piers $p_1 : AAAGTC$, $p_4 : AAACTT$ and $p_5 : AAAATG$ are inserted into the first bucket since the hash keys of these three piers are all $encode(AAA) = 000000$.*

5.2 Collision Handling

To handle collision caused by the insertion of piers into the same bucket of a hash table, the trivial solution is simply to store the piers in a bucket into a link list, named *collision list*, or consolidate them into an array to save space and increase accessing efficiency since the hash table structure is relatively stable once it is constructed. If the piers in the same bucket are stored as a list, each pier in the bucket will be retrieved and compared with the query pattern using dynamic programming. This will be inefficient when the number of piers in the candidate bucket is large. In our implementation, we choose λ to be 10, and thus it means that there are over 10^6 buckets in the hash table. Given 10^9 piers, each bucket contains 1000 piers on average. To minimize the computation cost of obtaining candidates for sequence similarity search, we propose to use global penalty matrix to handle the collision list of a hash table.

A $4^{2\omega}$ global penalty matrix (GPM), where $\omega = \ell_p - \lambda$, is built beforehand. All the possible segments in a DNA sequence with length ω are mapped into 2ω bit integers with the use of the encoded function described. We compute the edit distance for each pair of $\langle i, j \rangle$, where $0 \le i, j < 4^\omega$, and store the computed value into the cell $\langle i, j \rangle$ in the GPM. Note that by carefully and systematically eliminating symmetric cases, we can reduce the size of the table dramatically. When the piers in a candidate bucket need to be retrieved and checked during query processing, the edit distance of a pier and the query pattern does not have to be computed through dynamic programming. Instead, only the GPM is looked up to see whether the pier suffix segments and query pattern suffix is within a small edit distance ϵ. If it holds, we will say that the current pier p_i may be a candidate of the query pattern q_j since we know that the current pier and the query pattern share the similar prefix with length ℓ_p. By using the pre-computed GPM, the computation cost of verification can be reduced from $O(\ell_p^2)$ to $O(1)$.

6. QUERY PROCESSING

In this section, we will show how the hash-based pier model can be used to achieve efficient and effective similarity search in a biological sequence database. The space and time complexity are also discussed and analyzed. In our approach, the algorithm of sequence similarity search based on the hash-based pier model consists of three steps: generating the query pattern with size of ℓ_p from Q; searching for pier candidates among the hashed piers; and post-processing the candidates to concatenate adjacent candidates to form final alignments with a high alignment score. We focus our discussion on the second step since it is the main part of query processing.

6.1 Neighborhood Enumeration

Our search technique partitions the given query sequence Q into $|Q| - \ell_p + 1$ query patterns $q_1, q_2, ..., q_{|Q|-\ell_p+1}$. In the second step, the pier prefix segment with length λ, $q_i[0, \lambda-1]$ is encoded to a hash key h_i, which is a 2λ bit integer. Then all the encoded neighbors *ngbr* of this hash key h_i are enumerated, and the neighbors are those 2λ bit integers which are within a small edit distance from h_i. In our algorithm, for simplicity, λ is set as 10 or 12, and the edit distance allowed for neighbors is set as 2. Our method for enumerating all neighbors within an edit distance of 2 from the given encoded query pattern is supported by Theorem 6.1. Note that each neighbor of the hash key can be enumerated in $O(1)$ amortized cost.

THEOREM 6.1. *Let S and Q be two sequences of length λ from the alphabet set Σ. If $edit(S, Q) \le 2$, then one of the following cases is true:*

- *Case 1: $edit(S, Q) = 0$, i.e. the two sequences are exactly the same;*
- *Case 2: $edit(S, Q) = 1$, i.e. one replacement operation is needed in S to transform S to Q;*
- *Case 3: $edit(S, Q) = 2$, there will be three subcases to explain it:*
  - *Case 3.1 two replacement operations in S are needed to transform S to Q;*
  - *Case 3.2 one insertion and one deletion in S with order are needed to transform S to Q;*
  - *Case 3.3 one deletion and one insertion in S with order are required to transform S to Q.*

Based on Theorem 6.1, the encoded neighboring 2λ bit integers are enumerated. We shall show in detail how the neighbors of the given encoded sequence q of length λ are generated.

Case 1 means that the neighbor of q is q itself. In Case 2, the neighbor of q is enumerated with the replacement of the letter x in position i, $0 \leq i < \lambda$ with other letters in Σ. Each neighbor *ngbr* enumerated in terms of Case 2 meets $edit(q, ngbr)$=1. In Case 3.1, the neighbor is generated with the replacement of the letter x in position i, $0 \leq i < (\lambda - 1)$ with other letters except x, followed by the replacement of the letter y in position j, $(i+1) \leq j < \lambda$ with other letters except y. In Case 3.2, the neighbors are enumerated with the insertion of any letter in Σ in position $0 \leq i < (\lambda - 1)$ of q and the deletion of the letter in position j, $(i+1) \leq j < \lambda$ of q. Similarly, in Case 3.3, the neighbors are generated with the deletion of a letter in $0 \leq i < \lambda$ and the insertion of any letter in Σ in position j, $(i+1) \leq j < \lambda$ of q. Each neighbor *ngbr* enumerated in terms of Case 3.1, Case 3.2 and Case 3.3 meets $edit(q, ngbr)$=2.

There may exist several kinds of redundancy in neighbor enumeration, which means that the same neighbor may possibly be enumerated several times across the different cases in Theorem 6.1, or the neighbor generated may be the sequence itself.

Most of the redundancy can be avoided easily with little additional cost. For those redundancy that cannot be detected easily, we will exempt it by labeling those neighbors that have been enumerated already.

6.2 Sequence Similarity Search

The algorithm of sequence similarity search using the proposed hash-based pier model is presented in Algorithm 1. Once an encoded neighbor *ngbr* of the hash key of q_i is enumerated, the piers in the bucket of the hash structure $HTable[ngbr]$ will be retrieved and checked to see if they are candidates, i.e., whether they are similar to the query pattern q_i by the *verify* function. The *verify* function is implemented using the global penalty matrix (GPM) we mentioned in earlier section.

For a pier candidate, the edit distance is allowed twice, once when enumerating neighbors, and once when using the GPM. The two edit distances can not be simply added up as the edit distance between pier p and query pattern q_i. In order to ensure a high sensitivity, suppose that the edit distance for enumerating neighbors is β and the one for the GPM is ϵ. By careful setting of β and ϵ, we can capture most cases of $edit(p, q_i) \leq \theta$. Due to the space limitation, we will ignore the theoretical analysis here.

```
Algorithm 1: Hash-based Similarity Search Algorithm
Input: Hash table $HTable$, Query sequence $Q$, $\ell_p$, $\lambda$
Output: $Candidate$
Method:
begin
  $Candidate \leftarrow \emptyset$;
  for each query pattern $q_j$ in $Q$ do
    do enumerate the next neighbor $ngbr$ of $encode(q_j[0, \lambda])$;
      for each pier $p$ in the bucket $HTable[ngbr]$ do
        $verify(p, q_j)$;
        if $p$ and $q_j$ are similar then
          $Candidate \leftarrow Candidate \cup \{\langle p, q_j \rangle\}$;
    until all the neighbors of $q_j$ are enumerated;
end
```

6.3 Time and Space Complexity

Parameter	Values in Case I	Values in Case II
ℓ_p	15	18
ℓ_s	7	9
λ	10	12
θ	3	3

Table 2: The Parameters in the Experiment

We next look at the time and space complexity for our algorithm. First, for pier set P construction, we need to scan the database once; the time complexity for this is $O(|D|)$. A more effective way is to simply read the piers that we need since we can obtain the start position of each pier on the fly. To build a hash table for the piers, each pier can be inserted with time complexity $O(1)$ if the global penalty matrix is used. So the total time complexity for the construction of the hash table will be $O(|P|)$. For the space complexity of the hash table, we need $O(4^\lambda)$ for the table head. For the bucket of the table, each pier will contribute space $\Theta(\omega)$. Thus, the total size of the buckets will require space $O(\omega|P|)$. Also, we need space $\Theta(4^{2\omega})$ for the global penalty matrix. Thus, the total space complexity for the hash structure will be $O(4^\lambda + 4^{2\omega} + \omega|P|)$. Typically, if we set $\lambda = 10$ and $\omega = 5$, for a sequence with 3×10^9 letters, with span length $\ell_s = 7$, then the hash table size will be less than 200 mega-bytes. The structure is small enough to keep in the main memory to aid faster computation.

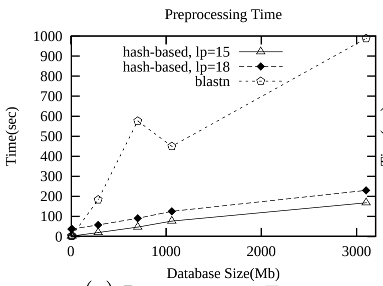
For each query pattern q of the query Q, the set of piers N of the neighbors for q will be enumerated. As we have explained, each neighboring pier can be generated with time amortized complexity $O(1)$. For each collision list we access, each item of the collision list will require time $O(1)$ when the global penalty matrix is used. Thus, the time complexity for the query is $O(\alpha|Q||N|)$, with the loading factor $\alpha = |P|/4^\lambda$. We can take the repeating computation into consideration, such as in symmetric cases, the letter repetition in a sequence and the neighboring of the serval query patterns, so as to reduce time complexity.

7. EXPERIMENTS

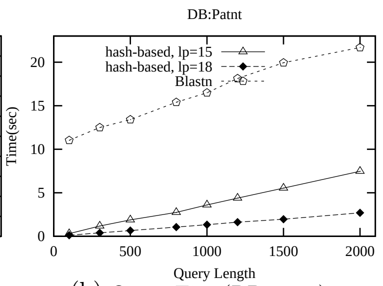
In this section, we present experimental results on the performance of homology search in DNA sequence databases to evaluate the efficiency of our proposed method for preprocessing and query processing in comparison to the latest version of BLAST (NCBI BLAST2).

7.1 Data sets

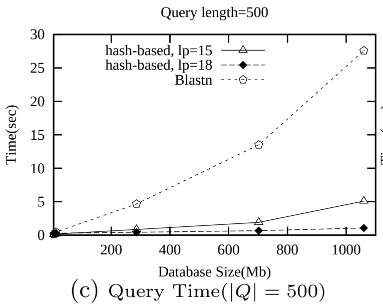
All the DNA sequence databases used in the experiment are downloaded from NCBI website. We use two sets of parameters shown in Table 2 for our method and set the seed length w=11 for BLAST. The DNA databases used in the experiments are the following: human_genomic: 3.1G, other_genomic: 1.06G, Patnt: 702.1M, month.gss: 286.2M, yeast.nt: 12.3M and ecoli.nt: 4.68M. We have collected a query set from a variety of sources to evaluate the responses from the hash-based pier approach and BLAST. The reported performance results are the average over 10 queries. The experiments are implemented in c programming language, and are executed on a Sunfire 4800 machine with 12 Ultrasparc3 CPU of 750MHz, 16GB free memory and 70GB free harddisk.



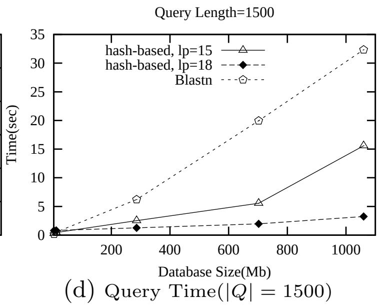
(a) Pre-processing Time



(b) Query Time(DB:patnt)



(c) Query Time($|Q| = 500$)



(d) Query Time($|Q| = 1500$)

Figure 3: Experimental Results for Efficiency

7.2 Performance Analysis

We perform several experiments to evaluate query processing in our hash-based pier model. Since sensitivity analysis has been done in our paper, we shall focus on showing the efficiency of query processing.

7.2.1 Efficiency in Pre-processing

We first perform an experiment that evaluates efficiency in data sequence pre-processing before performing sequence similarity search. Figure 3(a) shows that pre-processing with our method is much faster than with BLAST. This is because our hash-based pier model simply extracts piers and hashes them rather than processes each segment in the sequence database as BLAST and other methods do. Also the query pre-processing using the parameter values in set I is faster than using the ones in set II due to the features of hash tables for different parameter values.

7.2.2 Performance in Query Processing

An experiment is carried out to investigate how the length of the query sequence affects the performance of our method in comparison with BLAST. To do this, we perform similarity search for query lengths of 100, 300, 500, 800, 1000, 1500 and 2000 on database *patnt*. Figure 3(b) shows our search speed is 2-15 times faster than BLAST when the query length is varied.

For further evaluation of the efficiency of our method, we run two groups of queries with length 500 and 1500 on five data sets. As shown in Figure 3(c) and Figure 3(d), our method outperforms BLAST 2-10 times when the size of data sets varies for both groups of queries. This means that our method is very capable of handling sequence similarity search in large DNA sequence databases. The results also show that query processing using the parameter values in set II is about three times faster than using the ones in set I. This is reasonable since the number of candidates for $\ell_p = 18$ is smaller than $\ell_p = 15$ when $\theta = 3$ in both cases. Fewer candidates lead to lower computation cost, and therefore much faster speed.

8. CONCLUSION

In this paper, for efficient similarity search, we have proposed a new model – the hash-based pier model, and demonstrated its search efficiency and sensitivity. We have also proposed a method – the GPM-based method – to further improve search efficiency by reducing the computation cost of candidate verification. Compared to the most widely used biological database search tool, BLAST, our method is faster, yet requiring smaller memory and space.

9. REFERENCES

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. A basic local alignment search tool. In *Journal of Molecular Biology*, 1990.

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

[3] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. 1994 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'94)*, pages 419–429, Minneapolis, Minnesota, May 1994.

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

[5] D. Gusfield. *Algorithms on Strings, Trees and Sequences, Computer Science and Computation Biology.* Cambridge University Press, New York, 1997.

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

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

[8] B. Ma, J. Tromp, and M. Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[9] U. Manber and G. Myers. Suffix arrays: a new method for on-line string search. *SIAM Journal on Computing*, 22:935–948, 1993.

[10] 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.

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

[12] O. Ozturk and H. Ferhatosmanoglu. Effective indexing and filtering for similarity search in large biosequence datasbases. In *Third IEEE Symposium on BioInformatics and BioEngineering (BIBE'03)*, Bethesda, Maryland, 2003.

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

[14] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Molecular Biology*, 147:195–197, 1981.

[15] Z. Tan, X. Cao, B.C. Ooi, and A. Tung. The ed-tree: an index for large dna sequence databases. In *In Proc. 15th Int. Conf. on Scientific and Statistical Database Management*, pages 151–160, 2003.

[16] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th IEEE Symp. On Switching and Automata Theory*, pages 1–11, 1973.

[17] H.E. Williams and J.Zobel. Indexing and retrieval for genomic databases. *IEEE Transactions on Knowledge and Data Engineering*, 14:63–78, 2002.