Fast Dictionary Lookup in Genomic Information Retrieval

Simon M.C. Yuen, Fu-lai Chung[†] and Robert W.P. Luk

Department of Computing, Hong Kong Polytechnic University
Hung Hom, Kowloon, Hong Kong.

***Abstract*—Indexing and retrieval techniques for homology searching of genomic databases are increasingly important as the search tools are facing great challenges of rapid growth in sequence collection size. Consequently, the indexing and retrieval of possibly gigabytes sequences become expensive. In this paper, we present two new approaches for indexing genomic databases that can enhance the speed of indexing and retrieval. We show experimentally that the proposed methods can be more computationally efficient than the existing ones.**

Keywords: Homology Search, Genomic Database, Hashing, Trie.

I. INTRODUCTION

Information search and retrieval have been used in many areas such as internet, libraries, financial reports, and even biological data. Since these collections usually contain great amount of textual information, researchers have developed many similarity scoring schemes and search data structures to improve the efficiency and effectiveness in the last decades. Similarity scoring schemes are used to compute how similar the database items to the query based on certain characteristics. Data structures are used in search tools for managing a set of keywords and there are many kinds of data structures such as binary trees, splay trees, tries, burst tries and hash tables. In this paper, we focus on the latter one.

Genomic homology search tools can help biologists to identify related genes and functional groups. Generally, genomic search tools can be divided into two groups: exhaustive and index-based. Exhaustive search tools match queries to each sequence in the database [15]. Index-based search tools look up subsequences and their corresponding posting lists in some well-defined data structures. For example, FLASH [1], RAMDB [4], MAP [27] and CAFE [7-10] have adopted indexing techniques in their search tools. The advantage of index-based search tools over the exhaustive ones is that the pre-built indices can help to speed up the search process. CAFE claimed that it could run eight times faster than BLAST [12], [16], [21] and 50 times faster than FASTA [13].

There are three major considerations of using indexing techniques in genomic search tools: space requirement, sensitivity and effectiveness of retrieved sequences as well as the efficiency of indexing and retrieval. For example, CAFE addressed the first two problems by its compression techniques and two-component search processes while some researchers proposed some indexing techniques to enhance the efficiency [6], [10], [24], [25]. Our goal is to address the third consideration above, i.e., making index-based search tool more practical and scalable to increasing database size and query rates.

Data structures are used for managing a set of strings in the process of index construction and retrieval. The performance of data structure can significantly affect the query processing time. During the index construction process, a huge amount of distinct words in the sequence database are stored into a data structure. Since the genomic database size is doubling almost yearly and the search tools have to regularly rebuild their indices, the involved process becomes more and more expensive. Moreover, the great amount of queries would cause a heavy load to the servers. Many research papers proposed different kinds of search data structures focusing on document retrieval on the web or libraries. Here, we are interested in studying which types of data structures are best for searching biological sequences.

In this paper, we propose two new dictionary lookup approaches to minimize the cost of index construction and retrieval in genomic database search. Our proposed approaches can speed up the index-based tools such as CAFE, which are more than 5 times faster than the bitwise hashing and 7 times faster than the burst trie. We show experimentally our two proposed data structures running much faster than other data structure for the task of looking up genomic sequences. We also investigate the effect of the database size and the word size to the performance of data structures.

II. DATA STRUCTURE FOR DICTIONARY LOOKUP

Most index-based search tools choose fixed-length intervals rather than variable-length due to better effectiveness. In general, during the index construction, each distinct interval is dynamically added to a data structure, together with its

[†] *Corresponding author: cskchung@comp.polyu.edu.hk*

frequency and sequence id. The data structure is usually stored in main memory to avoid disk I/O time. There are many kinds of data structures that can be used to manage these intervals including hash-tables, trees, tries, burst tries, etc. We would like give a brief review of these structures.

A. Hash Tables

A hash table stores records by using a hash function, which maps a string key to a bucket. The naïve hashing function is done by linear probing, but it is not efficient enough. There are many different kinds of hashing functions [5], [19], [26], some of them are complicated and used for security encryption. However, for the task of string searching, we need a hashing function using efficient and less mathematical operations such that the process of vocabulary accumulation can be done quickly. Two fast hashing functions are commonly used, namely, Bitwise and Perfect Hashing.

1) Bitwise Hashing Function

Zobel proposed the bit-wise hashing function [30], which can use a non-prime number as its seed. The advantage of bit-wise hashing function is that the bucket size is the power of 2. Typically, most hashing functions require the complex mathematical operations such as power and module. The bit-wise hashing function adopts shift bit operations rather than computationally heavy operations. It is claimed that the bit-wise hashing function is 10-30 times faster than uniform string hashing function.

2) Perfect Hashing Function

The characteristic of perfect hashing is that no collision occurs for any different words and the lookup time is guaranteed to be O(1) even in the worst case. Each unique word is assigned to a unique hashing integer. The bucket size is the number of all possible words (independent of database size). For example, if the word size of nucleotide sequences is 9, the bucket size is 4^9.

A word is assigned with a unique hash value according to the orders of its predefined alphabetical ID in the word. The calculation of a hashing integer can be done by a formula. The parameters of the formula are the alphabet ID and the position of each character. We assume that the leftmost character in a string is most significant while the rightmost character is least significant. We can calculate the hash value by an ordered function. For a string "$a_{n-1}a_{n-2}\ldots a_1a_0$",

$$\textit{Hash value of the string} = \sum_{\forall k} P_k \times L^k$$

where P_k is the alphabet ID of the k-th character in the string and L is the number of alphabet characters.

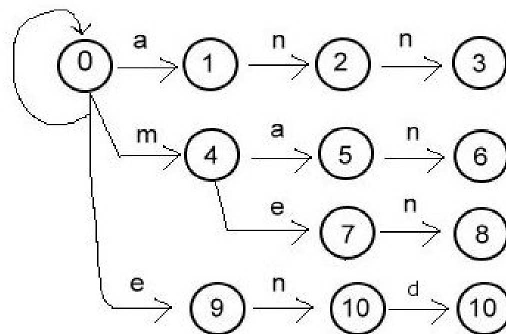
B. Trie

Trie is an efficient data structure to identify a set of keywords in a string efficiently [11], [20], [23], [28], [29]. It has been used in various applications like library bibliographic search, natural language dictionaries, database systems, compliers and trie image. Trie can be divided into two parts [14]. The first part is to construct a finite state matching machine based on the input set of keywords. Its creation process takes the time proportional to the length of keywords. The second part is to process an input text so that the finite state machine signals the matching keywords. The advantage is that the running time of processing a text is independent of the number of keywords.

The pre-built pattern matching machine processes a text and then reports the matching signals. The machine consists of a number of states represented by consecutive integers. When the characters are read from the text one by one, the machine will move from one state to the other state and this is called state transitions. The state transitions and the matching signals are monitored by three functions: *Goto* function, *Failure* function and *Output* function. These functions for the keywords {ann, man, men, end} are demonstrated in Fig 1.

The finite state machine processes the text and the characters are examined one by one. If a character is matched with the symbol in the existing state, the Goto function is used and the machine will move to the next state. The Failure function is used to provide the exit state if failure in matching occurs. The output function gives the signal that a keyword is found in the text string.

The advantage of using a trie is that the state machine can look up a query string in a set of keywords and the cost is independent to the number of keywords. Trie is fast because it requires only one transition for processing each letter in the query string.



(a) Goto function

State	Output
3	ann
6	man
8	men
12	end

(b) Output function

State	Failure
6	2
8	10

(c) Failure function

Fig.1 Pattern Matching Machine in Trie

Brain and Tharp also proposed a two-dimensional array-based trie [17], which stores the entry id of a minimal word list. Take the example of finding the entry id of a word

"apple" in the word list. First, look at the first column and the row 'a'. If the cell value is positive, we get the entry id and stop searching. If the cell value is negative, that means there are more than one words with first character 'a'. Then, continue to look at the second column and the row 'p' and check again the cell value. If it is negative, repeat the steps. The worst retrieval time is O(n), where n is the maximum word length. The table size depends on the maximum word length. The table can be placed in main memory to avoid expensive disk access. However, this approach is only practical at searching a small amount of words. The maximum number of entries in the table is L×n, where L is the number of alphabets and n is the maximum word length. This approach is not feasible for genomic sequences because the possible combinations of words in a database are much more than the size of that table.

C. *Search Trees*

A search tree starts with a root node and each node stores an element and points to the node's children [22]. A popular search tree is the Binary Search Tree (BST), which is a binary tree where every node's left sub-tree has keys less than the node's key and every right sub-tree has keys greater the node's key. The performance of a BST is proportional to the height of the tree. The average searching time is O(log n). But in the worst case, the searching time is O(n). There are variants of search trees such as: AVL trees, red-black trees and splay trees. In our experiments, we only test BST since it is shown that BST is faster than other search trees and has nearly same performance as red-black trees in vocabulary accumulation.

III. Proposed Data Structure for Genomic Sequences

Although there are many different data structures for vocabulary accumulation of words, few of them are efficient enough for genomic sequences. Genomic sequences have some special characteristics. They are very long and nearly random as well as they are usually broken into fixed size of words for indexing. Although the bit-wise hashing and the perfect hashing functions are supposed as very efficient data structures for indexing, we observe that the performance of all these data structures decreases with word size. For example, the perfect and the bit-wise functions require more time to calculate for longer words, while trie requires more state transition to go through more letters. We consider that the performance can be significantly improved by a simple heuristics, i.e., utilizing the overlapping characteristic in words to speed up the hashing functions. We propose two new data structures for genomic sequences that can outperform other data structures and guarantee the running time be O(1).

A. *Refined Perfect Hashing Function*

The idea of the refined version is to calculate a new hash value of a word based on that of the previous consecutive word. Therefore, the number of operations required in refined version can be much reduced. For example, a new sequence ACTAAAAAAGTTGACG is indexed. The sequence is broken down into words of size 9: ACTAAAAAA, CTAAAAAAG, TAAAAAAGT, etc. The hash value (24576) of the first word is calculated by the formula in the perfect hashing function. Based on the first word, the second word requires fewer operations by just dropping a character 'A' and adding a 'G'. The steps of our refined version to calculate the new hash value are shown in Fig 2 and they can be exemplified below.

- Hash value of CTAAAAAAG

  $= (24576 - 0) << 2 + 3 = 98307$

- Hash value of TAAAAAAGT

  $= (98307 - 4^8) << 2 + 2 = 131086$

1. Use the hash value of previous word
2. Minus the value of the missing character
3. Perform a left shift operation
4. Add the value of the last character

Fig.2. Steps to compute hash values of the refined perfect hashing

B. *Trie-based Hashing Function*

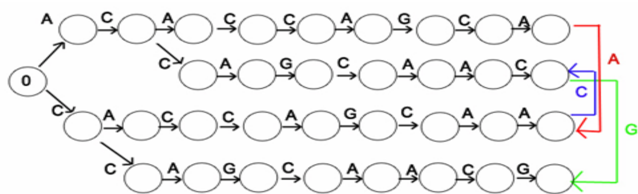
As mentioned before, the finite matching machine of a trie is efficient in matching a query string to a set of keywords. The trie-based hashing function can return unique hash value for each distinct word. Like other hash tables, the hash value generated from trie-based hashing function represents an address of the corresponding posting record. Our proposed trie-based hashing function is efficient, as compared with the bit-wise and other trie-based hashing functions in our experiments. We introduce the concept of loop back path in the trie for managing the overlapping intervals. First, we would explain why our proposed hashing functions can be faster in search sequences. Secondly, we show two different implementations of the trie-based hashing function.

1) *Loop Back Path in Trie*

A machine state is built by pre-processing the database sequences. The pre-calculated states are stored in the three functions. Since the consecutive words are overlapping, we can add a direct path to a state for the next word, which is called loop back path here. Loop back path can help to reduce the number of state transitions in processing a series of consecutive words. For example, a sequence ACACCAGCAACG is broken into a set of words

Sequence	SID
ACACCAGCA	100
CACCAGCAA	101
ACCAGCAAC	102
CCAGCAACG	103

with given unique hash values called SID. The state machine for these words with SIDs is shown as follows.



(a) Goto_function

Sequence ID	Failure State
100	102 if Next Character is A
101	103 if Next Character is G
102	101 if Next Character is C
103	NIL

(b) Failure function

Fig.3. Pattern Matching Machine for Trie-based Hashing with Loop Back Path

The state machine follows a loop back path to access the next overlapping word. The modified failure function is to provide a failure state according to the next character. It is fast to get a hash value since only one state transition is needed. The loop back paths enable the machine to skip the process of duplicated characters. With the loop back paths, the machine does not require to restart the state from 0 after processing every word. Therefore, it can save much more operations for indexing billions of long sequences.

The performance of the trie-based hashing function is independent of the word size. When the state machine searches a genomic query sequence of length X, it first looks up the first word in the sequence by the trie structure. Then, it utilizes the loop back paths to obtain the hash values for the remaining X-N+1 words. Thus, it only requires X-N+1 state transitions. Since the state transitions are recorded in the failure function, the in-memory operations can be done very fast. Table I shows the number of operations required to lookup a word of size N in the worst case.

Table I. The number of operations required for processing a consecutive N-gram: Worst case scenarios

	Add Operation	Shift-Bit Operation	Multiple Operation	XOR Operation	Module Operation	Transition Operation
Perfect Hash	N	N(N-1)/ 2	N	0	0	0
Bitwise Hash	2N	2N	0	N	1	0
BST	0	0	0	0	0	N
Refined Perfect	2	N	1	0	0	0
Trie with Loopback	0	0	0	0	0	1

C. Implementation

There are two ways to implement the loop back paths in trie. A trie hashing can form a minimal or a non-minimal item list. The first one requires going through the database once to build the output and failure functions while the second one statically generates the functions.

In the first implementation, all the distinct words in the database are extracted and stored in a trie structure. The leaf node of each path is a unique hash value SID. The number of records is equal to the number of unique words in database. Fig.4 shows the first implementation of the trie-based hashing function for nucleotide sequences. There are two main components, namely, a trie structure and a failure function table. First, each word has a hash value, which is stored in a trie-structure. We implement the trie-structure with a tree because the time of insertion, deletion and lookup of items is constant and independent of the tree size. Second, the failure function that stores the loop back path pointers for next characters is implemented by a 2-dimensional array table. Since there are 4 possible characters in nucleotide sequences, there are 4 possible loop back pointers in each row.

In this approach, all the sequences are examined and the distinct words are inserted into the trie. The time of building the trie and the failure function is one-time off. Since the words that do not exist in the database would not be stored in the array, the space requirement is proportional to the number of distinct words. This approach might be preferred for a small and highly similar sequence collection, where the number of distinct words could be small.

Sequence	SID
ACACAA	101
ACAATC	102
ACAATG	103
CACAAT	104

(a) Sequences

SID	A	C	T	G
101	-	-	-	104
102	-	-	-	-
103	-	-	-	-
104	-	102	-	103

(b) 2-dimentional array for Failure Function

Fig.4. First Implementation of Trie-based Hashing

In the second implementation, all the possible n-grams are assigned with a unique hash value as shown in Fig.5. The hash value is equal to that calculated by the Perfect hashing function. Instead of using a trie, we use the Perfect hashing function to lookup the first word in a sequence. Then, we find the remaining consecutive sequences by the failure hashing function. Unlike the first implementation, the failure hashing function returns the loop back pointers for the character A and the other three pointers are calculated. Thus, the function is implemented by an array of size L^n, where L is the number of alphabet and n is word size. For example, the machine has just finished GACACA and then processes ACACAG. The hash value can be obtained by the pointer value 272 plus 3 = 275.

A
0
1100
3140
272

(a) Sequences

Sequence	SID
AAAAAA	0
ACACAG	275
AGACAC	785
GACACA	3140

(b) An array for Failure Function

Fig.5. Second Implementation of Trie-based Hashing

We found that it is easier to implement it, in which the loop back pointer values could be calculated without reading any genomic sequences. This hash table might be larger than that in the first implementation if the number of distinct words in the database is small. The building time of the failure function is less than that of the first implementation. The time is bounded by O(n) or the size of table. Generally, the building time would affect the whole indexing process slightly because the number of words in a genomic database is usually much larger than the table size.

IV. EXPERIMENTAL RESULTS

In order to study the efficiency of the proposed data structures for genomic sequences, we have conducted two experiments. The first one is to measure the time required to perform dictionary lookup for a large amount of genomic sequences. The second one is to evaluate the space required for the proposed two trie-based hashing.

A. Test Data

In our experiments, a subset of genomic database GenBank is used. We perform dictionary lookup on the n-grams in a genomic sequence file containing Expressed Sequence Tag. Our proposed methods were written in C in Unix environment and the CPU time of running the hashing functions to index the nucleotide sequences was recorded.

B. Indexing Efficiency

Fig. 6 shows the running time in clock cycles for each data structure on the data set, for the task of vocabulary lookup for a genomic sequence file in GenBank. We have run on several data structures including Binary Search Tree (BST), Burst Trie, Bit-wise Hashing, Perfect Hashing, the proposed Refined Perfect Hashing and Trie-based Hashing A1 and A2. In this experiment, we use different n-gram sizes, i.e., 3, 7 and 10, Sequence Data File: gbest10.seq, Sequence Length: 100-1000, File Size: 28.6 MB.

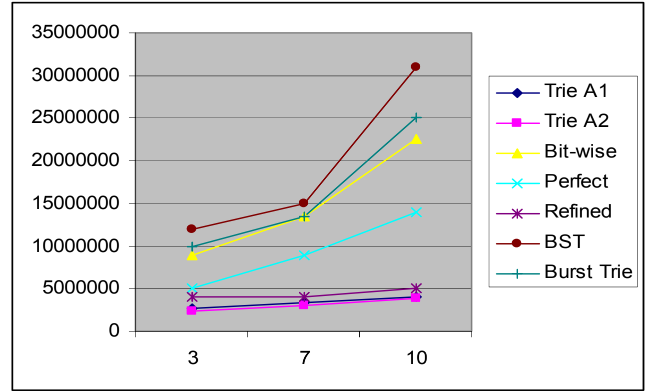


Fig.6. Plot of clock cycles required by several data structures, for the task of vocabulary lookup.

Our experiment results show that the time required for dictionary lookup increases with the word size. Among all data structures, the Trie-based Hashing using the second implementation, i.e. A2, is the most efficient for the genomic sequence. Our proposed Trie-based Hashing A1&A2 outperform other data structures and they are about 5 times faster than the Bitwise Hashing and about 4 times faster than the Perfect Hashing. The refined Perfect Hashing function is nearly as efficient as the Trie-based Hashing. The proposed Trie-based and refined Perfect Hashing run efficiently and the time required only slightly increases with word size. The result proves that these approaches are independent of word size. Burst Trie and Binary Search Tree are relatively slow since they require more comparisons and state transitions.

C. Space

The space requirement is one of the main considerations of the feasibility of the indexing techniques. Most indexing techniques speed up the searching process by allocating space for pre-built indices. But, the Bitwise, Perfect and Refined Perfect hashing functions, calculating the hashing value of genomic words on the fly, do not need any space. Our proposed trie-based hashing also requires memory space to store the failure function table. In the following, we briefly explain the space requirement for the two implementations of trie-based hashing.

In the second implementation of trie-based hashing, an index array table is created to store the loop back pointers. Thus,

Space size of index table = $L^n \times$ (size of a pointer).

On the other hand in the first implementation, the number of alphabet characters L might be set to a larger number. It is because if L is a power of 2, then the hash values can be calculated by using the inexpensive shift operations. For example, the number of alphabet character in amino acid is 20 and then we should set L as 32. This will increase the table size and more space is required.

On the contrary, the table size of the second implementation is merely affected by the unique words in genomic database and is bounded by 20^n. We found that the number of distinct words in database is related to the utilization which is defined as the number of unique words in a genomic database to that of a

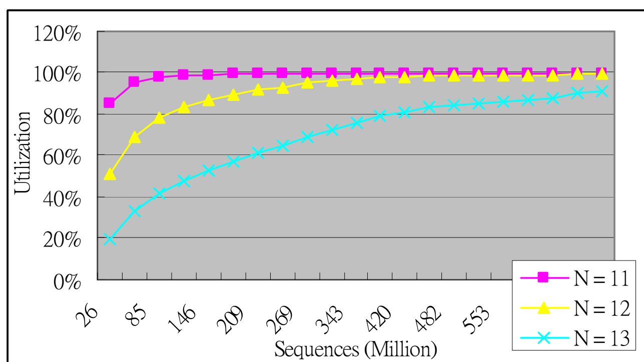
perfect table, i.e.

$$Utilization = \frac{No.\ of\ distinct\ n\text{-}grams\ of\ a\ genomic\ database}{No.\ of\ all\ possible\ n\text{-}gram}.$$

We are interested in which type of implementation is suitable for genomic database. The utilization of the genomic database is affected by the size of n-gram, the genomic database size, and the similarity of sequences. Low utilization means that the hash table of the first implementation would be smaller than the second one. In this experiment, our aim is to discover the utilization of a genomic database. The discovered utilization can implicitly reflect the distribution of the genomic database. We test with different sequence bases and word sizes.

Fig.7. The utilization of gbest.seq for n=11, 12 and 13

Based on the above result, the utilization steadily increases with the sequence bases. Since the number of possible n-grams is significantly affected by the word size, the utilization obviously drops when using larger n-gram. However, the utilization will reach 100% when more sequences are inserted. This result shows that almost all possible n-grams exist in genomic database when the amount of genomic sequences is large enough. It means that for a nucleotide database with



small n-gram, the second implementation should be preferred because the space requirement of the failure function is nearly the same as the first implementation, but the second one can run faster.

V. Conclusions

Genomic analysis is very crucial to the discovery of evolutionary relationship. Since the genomic database size is greatly increasing, it is expected more search tools will adopt indexing techniques in query processing. But the disadvantages of index-based search tools are the high cost of index construction and heavy load of the server. In this paper, we propose two new dictionary lookup approaches for genomic sequences that can improve both the index construction and retrieval processes.

Experimental results show that the proposed Refined Perfect hashing and the Trie-based hashing functions are much faster than the existing data structures. Data structure is a key component in vocabulary searching and the performance of data structure affects the efficiency of search tools. We believe that the proposed data structure can improve the genomic homology search. In our future work, we will study the correlation of sequences with the pre-built index table and the loop back pointers. With the knowledge of correlation of sequences, we can cluster the homology sequences and enable searching within clusters.

Acknowledgement

This work is supported by the Hong Kong Polytechnic University Grant under projects A-PE36 and Z08R.

References

1. A. Califano and I. Rigoutsos, “FLASH: A fast look-up algorithm for string homology,” *IEEE Computational Science & Engineering*, vol.1, no.2, pp.60-75, 1994.
2. A. Chattaraj and H.E. Willams, “Variable-length intervals in homology search,” *Proc. Second Asia-Pacific Bioinformatics Conference* (*APBC2004*), Dunedin, New Zealand, pp.85-91, 2004.
3. A. Chattaraj, H. E. Williams and A. Cannane, “General purpose search techniques for genomic text,” *Genome Informatics*, vol.15, no.2, pp.42-51, 2004.
4. 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 Biosciences*, vol.11, no.3, pp.273-279, 1995.
5. C. Silverstein, “A practical perfect hashing algorithm,” DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol.59, pp.23-47, 2002.
6. E. Hunt, M. P. Atkinson and R. W. Irving, “Database indexing for large DNA and protein sequence collections,” *VLDB Journal*, vol.11, no.3, 2002.
7. H. E. Williams, “Genomic information retrieval,” *Australasian Database Conference on Database technologies*, Adelaide, Australia, pp.27-35, 2003.
8. H. E. Williams and J. Zobel, “Indexing nucleotide databases for fast query evaluation,” *Proc. Int’l Conf. Advances in Database Technology* (*EDBT*), pp.275-288, 1996.
9. H. E. Williams and J. Zobel, “Indexing and retrieval for genomic databases,” *IEEE Trans. on Knowledge and Data Engineering*, vol.14, no.1, pp.63-78, 2002.
10. H. E. Williams, J. Zobel and P. Anderson, “What’s Next? Index Structures for Efficient Phrase Querying,” *Australasian Database Conference*, pp.141-152, 1999.
11. J. Aoe al et., “A Trie compaction algorithm for a large set of keys,” *IEEE Trans on Knowledge and Data Engineering*, vol.8, no.3, pp.476-491, 1996.
12. K. Ian, Y. Mark and B. Joseph, *BLAST*. O’Reilly and Associates, Dunedin, 2003.
13. L. Delcher al et., “Alignment of whole genomes,” *Nucleic Acids Research*, vol.27, no.11, pp.2369-2376, 1999.
14. L. Shang and T.H. Merretal, “Trie for approximate for string matching,” *IEEE Trans. on Knowledge and Data Engineering*, vol.8, no.4, pp.540-547, 1996.
15. M. Bin, J. Tromp and M. Li, “PatternHunter: Faster and more sensitive homology search,” *Bioinformatics*, vol.18, no.3, pp.440-445, 2002.
16. M. Cameron, H. Williams and A. Cannane, “Improved gapped alignment in BLAST,” *IEEE Trans on Computational Biology and Bioinformatics*, vol.1, no.3, 2004.

17. M. Brain and A. Tharp, "Using Tries to eliminate pattern collisions in perfect hashing," *IEEE Trans on Knowledge and Data Engineering*, vol.6, no.2, pp.239-247, 1994.
18. P. Baldi and S. Brunk. *Bioinformatics, The Machine Learning Approach*. The MIT Press, 2001.
19. R. J. Enbody and H. C. Du. "Dynamic hashing schemes," *ACM Computing Surveys*, vol.20, no.2, 1998.
20. R. Sinha and J. Zobel, "Cache-conscious sorting of large sets of strings with dynamic tries," *Journal of Experimental Algorithmics*, vol.9, 2004.
21. S. F. Altschul, T. L. Madden and A.A. Schaffer, "Grapped BLAST and PSI-BLAST: A new generation of protein database search programs," *Nucleic Acids Research*, vol.25, no.17, pp.3389-3402, 1997.
22. S. Burkhardt, A. Crauser and P. Ferragina, "q-gram based database searching using a suffix array (QUASAR)," *Proc. of the Third Annual International Conference on Computational Molecular Biology*, pp.77-83, 1999.
23. S. Ranjan and Z. Justin, "Efficient Trie-based sorting of large sets of strings," *Australasian Computer Science Conference*, Adelaide, Australia, pp.11-18, 2003.
24. S. Heinz and J. Zobel, "Performance of data structure for small sets of strings," *Proc. of the Australasian conference on Computer Science*, vol.4, pp.87-94, 2002.
25. S. Heinz and J. Zobel, "Efficient single-pass index construction for text databases," Journal of the American Society for Information Science and Technology, vol.54, no.8 , Pages 713 – 729, 2003.
26. S. Heinz, J. Zobel and H. E. Williams, "Burst tries: A fast, efficient data structure for string keys," *ACM Trans. Inf. Syst*, vol.20, no.2, pp.192-223, 2002.
27. T. Kahveci and K. Tamper, "Map: Searching large genome databases," *The Pacific Symposium on Biocomputing Conference*, Hawaii, pp.11-18, 2003.
28. W. A. Burkhard, "Hashing and Trie algorithms for partial match retrieval," *ACM Trans. Database Syst.*, vol.1, no.2, pp.175-187, 1976.
29. V. Alfred and J. Margaret, "Efficient string matching an aid to bibliographic search," Bell Laboratories, 1975.
30. M. V. Ramakrishna and J. Zobel, "Performance in practice of string hashing functions," in R. Topor and K. Tanaka, editors, Proc. Int. Conf. on Database Systems for Advanced Applications, pp.215-223, Melbourne, Australia, April 1997.