Indexing Huge Genome Sequences for Solving Various Problems

Kunihiko Sadakane[1] **Tetsuo Shibuya**[2]
sada@dais.is.tohoku.ac.jp tshibuya@trl.ibm.co.jp

[1] Graduate School of Information Sciences, Tohoku University, Aoba-yama09, Sendai 980-8579, Japan
[2] IBM Tokyo Research Laboratory, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan

Abstract

Because of the increase in the size of genome sequence databases, the importance of indexing the sequences for fast queries grows. Suffix trees and suffix arrays are used for simple queries. However these are not suitable for complicated queries from huge amount of sequences because the indices are stored in disk which has slow access speed.

We propose storing the indices in memory in a compressed form. We use the compressed suffix array. It compactly stores the suffix array at the cost of theoretically a small slowdown in access speed. We experimentally show that the overhead of using the compressed suffix array is reasonable in practice. We also propose an approximate string matching algorithm which is suitable for the compressed suffix array. Furthermore, we have constructed the compressed suffix array of the whole human genome. Because its size is about 2G bytes, a workstation can handle the search index for the whole data in main memory, which will accelerate the speed of solving various problems in genome informatics.

Keywords: DNA, similarity search, suffix array, compression

1 Introduction

The size of genome sequence databases is increasing. More than fifteen billion base pairs are stored in a database. Therefore we should construct indices for fast queries from the database. Because a genome sequence can be regarded as a text string, we can use various algorithms and data structures on strings in the literature. Many biological problems can be solved efficiently by using the algorithms on strings, for example sequence similarity search [13], extracting structured motifs [12], sequence assembly by using approximation algorithm for the shortest superstring [1], computing alignment of whole genomes [3] sequence database classification [6], etc. These applications use the suffix tree [15] or its space-economical alternative, the suffix array [11]. They are typical full-text indices in information retrieval. It is however difficult to use them for genome-scale databases because of their huge amount of index sizes. For example, even the smallest representation of a suffix tree of Kurtz [10], it is conjectured that the suffix tree will occupy 45G bytes of memory for the whole human genome of three billion base pairs. Though this seems to be reasonable size to store it in external disk, it is not suitable to solve the above problems because of the following reason: Unlike a simple pattern matching, the algorithms find the positions of occurrences of many patterns at a time, and calculate the relationship between the occurrences. Thus there is no locality of reference in disk accesses, which causes a slowdown in the calculation. Though some I/O efficient suffix trees have been proposed [2, 4, 9], these are concentrating on finding occurrences of a pattern. As the result, the above problems can be solved efficiently for only small amount of data so that its index fits in main memory.

Our approach to overcome the inefficiency of using indices for large data is to store them in main memory in a compressed form. Very recently some compression techniques for suffix arrays called *compressed suffix arrays* were proposed [8, 5, 14, 7]. By using these techniques the suffix array of a text string can be compressed in the size of the same order as the string itself at the cost of a small increase in access time. However this slowdown is much less than that by storing the indices in disk.

We have constructed the compressed suffix array for the whole human genome (draft assembly) of about 2.7 billion base pairs. Because the size of the index is less than 2G bytes, it will fit in main memory of a workstation, or even a personal computer. Various algorithms can be executed quickly using this index. As an example, we show experimental results of similarity search in genome sequences by using an approximate matching algorithm of Navarro and Baeza-Yates [13]. Because it uses the suffix array, we can also execute it on the compressed suffix array. We found that the approximate matching algorithm using compressed suffix array is at most twenty times slower than that using uncompressed suffix array. The size of the compressed suffix array is five times smaller than the uncompressed suffix array. Therefore the algorithm using the compressed suffix array is faster if the index fits in main memory, whereas the suffix array does not fit. We also propose an approximate matching algorithm which is suitable for the compressed suffix array. It shows a potential ability of the compressed suffix array.

2 Compressed Suffix Arrays

The compressed suffix array is an indexing data structure for text strings. Because it has the same function as the suffix array, we first explain the suffix array.

2.1 Suffix Arrays

We call a genome sequence stored in database a *text*. Let $T[1..n] = T[1]T[2]\cdots T[n]$ be a text of length n on an alphabet Σ. We assume that $T[n] = \$$ is a unique terminator that is smaller than any other symbol. For DNA sequences, $\Sigma = \{\$, \mathrm{a}, \mathrm{c}, \mathrm{g}, \mathrm{t}\}$. The j-th suffix of T is defined as $T[j..n] = T[j]T[j+1]\ldots T[n]$ and expressed by T_j. A substring $T[1..j]$ is called a prefix of T. The suffix array $SA[1..n]$ of T is an array of integers j that represent suffixes T_j. The integers are sorted in lexicographic order of the corresponding suffixes.

Any pattern P of length m can be found from the text in $O(m \log n)$ time, and the positions of all *occ* occurrences of the pattern can be enumerated in $O(m \log n + occ)$ time. It can be improved to $O(m + \log n + occ)$ time by using an auxiliary array called LCP array. Note that the LCP array is seldom used because its size is the same as the suffix array.

The size of the array SA is $n \log_2 n$ bits because $\log_2 n$ bits are necessary to represent a number in range $[1, n]$. Therefore the size of the suffix array is $n \log_2 n$ bits plus text size. Assume that the length of the genome sequence is at most $2^{32} = 4$G bases. Then SA can be stored in $32n$ bits, or $4n$ bytes. Therefore $12G$ bytes memory is necessary to store the suffix array of the whole human genome of 3 billion base pairs.

2.2 Compressed Suffix Arrays

The compressed suffix array is a compressed version of the suffix array. It has three basic functions: *lookup*, *inverse* and *decode*. *lookup*(i) returns $SA[i]$ in $O(\log^{\epsilon} n)$, *inverse*(j) returns $SA^{-1}[j]$ in $O(\log^{\epsilon} n)$, and *decode*(j, l) returns a substring $T[j..j+l-1]$ of length l in $O(l + \log^{\epsilon} n)$ where ϵ is any fixed constant in $0 < \epsilon \leq 1$. Because the size of the compressed suffix array is nearly proportional to $1/\epsilon$, we use $\epsilon = 1$ to reduce the size. To execute *decode*(j, l), it is not necessary to use an uncompressed text T. Therefore we can perform any operation using the suffix array and the text by using only the

lookup(i):	*inverse*(i):
$v := 0$;	$p := (i-1)/D$; $q := J[p]$; $v := 1$;
while $i \bmod D \neq 0$ do	while $p \cdot D + v < i$ do
$i := \Psi[i]$;	$q := \Psi[q]$;
$v := v + 1$;	$v := v + 1$;
return $I[i/D] - v$;	return q;

Figure 1: Functions $lookup(i) = SA[i]$ and $inverse(i) = SA^{-1}[i]$

compressed suffix array although it is slower by $O(\log n)$ factor. If we have an uncompressed text T, then we can execute $decode(j, l)$ in $O(l)$ time.

The compressed suffix array of the text T stores Ψ function defined as

$$\Psi[i] = SA^{-1}[SA[i] + 1] \text{ unless } SA[i] = n$$

instead of the suffix array SA. Each element of the suffix array can be computed in $O(\log^{\epsilon} n)$ time. The size of the compressed suffix array is $O(n)$ bits. Specifically, the size is $\frac{1}{\epsilon}(nH_1 + O(n))$ bits where H_1 is the order-1 entropy of the text. Therefore the compressed suffix array can be smaller than the original text if ϵ is large. Moreover, the compressed suffix array can be used without the text itself [14], that is, any pattern in the text can be found by using only the compressed suffix array, and any substring of the text of length l can be extracted in $O(l + \log^{\epsilon} n)$ time. This means that we can find any pattern from a compressed text.

In a practical implementation of the compressed suffix array [7], ϵ is set to 1. The data structure consists of the following components:

- $\Psi[i]$ for $1 \leq i \leq n$
- $I[i] = SA[i \cdot D]$ for $1 \leq i \leq n/D$
- $J[i] = SA^{-1}[i \cdot D + 1]$ for $0 \leq i \leq (n-1)/D$
- $C[c]$ for $c \in \Sigma$ where $C[c]$ is the number of occurrences of characters in the alphabet Σ which are smaller than c

where D is a constant. The larger D, the less space and the less access speed. By using the above data structure an element $SA[i]$ can be computed by the pseudo code in Figure 1. Note that $\Psi[i]$ values are stored explicitly only if i is a multiple of a constant L and other values are represented as the difference from the previous value. We also set $L = O(\log n)$. We can compute any $\Psi[i]$ value in constant time by using table-lookups. We set $D = O(\log n)$. Then these function takes $O(\log n)$ expected time [7]. Therefore we can perform a binary search on the suffix array to search for a pattern P using the lookup function.

We can also compute the inverse function of *lookup*. This function is necessary to remove the text from the database. A character $T[i]$ can be found without using the text as follows. First we compute the lexicographic order p of the suffix $T_i = T[i]T[i+1]..T[n]$ by $p = inverse(i)$. Then we perform a binary search on the array C to find the first character of the suffix T_i. Because the first characters of suffixes $T[SA[j]]$ are sorted in alphabetic order and the indices of the boundaries between different characters are stored in the array C, we can compute $T[i]$ correctly.

3 Pattern Matching Algorithms using the Compressed Suffix Arrays

3.1 Backward Search

A drawback of the conventional binary search using the compressed suffix array is the time complexity to simulate a traversal on a suffix tree. If we are given the node of a suffix tree representing a

bsearch(l, r, c):
  $\hat{l} := C[c]$; $\hat{r} := C[c+1] - 1$;
  find the smallest l' such that $\hat{l} \leq l' \leq \hat{r}$ and $\Psi[l'] \geq l$;
  find the largest r' such that $\hat{l} \leq l' \leq \hat{r}$ and $\Psi[r'] \leq r$;
  return l', r';

Figure 2: Function *bsearch*(i)

pattern of length m $P = P[1]P[2]\ldots P[m]$, the node representing a pattern of length $m+1$ $Pc = P[1]P[2]\ldots P[m]c$ can be found in constant time using hash, or in $O(\log|\Sigma|)$ time using a binary tree. The corresponding operation using the suffix array is to compute the range $[l, r]$ of the lexicographic orders of suffixes. Let $[l_m, r_m]$ be the range of the suffix array corresponding to the pattern P. That is, suffixes T_j for $j = SA[l_m], SA[l_m+1], \ldots, SA[r_m]$ have the pattern P as their prefixes. Then we compute the range $[l_{m+1}, r_{m+1}]$ corresponding to Pc. This operation takes $O(m \log n)$ time by the conventional binary search on the compressed suffix array. Therefore it takes $O(m^2 \log n)$ time to calculate the ranges of the suffix array $[l_1, r_1], [l_2, r_2], \ldots, [l_m, r_m]$, each of which corresponds to prefixes $P[1], P[1..2], \ldots, P[1..m]$ of P. This means that it is not suitable for dynamic programming algorithms for approximate string matching.

We use another algorithm, called *backward search*, to find the range of the suffix array corresponding to P. Originally it has been proposed for the FM-index [5]. We can use a similar algorithm for the compressed suffix array. Unlike the conventional binary search, backward search iteratively finds suffixes of P from right to left. Figure 2 shows the function *bsearch*(l, r, c). Given the range $[l, r]$ of the suffix array corresponding to P, it computes a new range corresponding to cP, a concatenation of the character c and the pattern P. The function takes $O(\log n)$ time. Therefore we can enumerate m ranges corresponding to suffixes of a pattern of length m in $O(m \log n)$ time.

3.2 An Approximate Matching Algorithm

We propose an approximate string matching algorithm using the compressed suffix array. We first propose a simple algorithm that is based on the backward search we described in the last section, and then extend it using a technique by Navarro and Baeza-Yates [13]. As a similarity measure, both algorithms can deal with either of edit distance or ordinary alignment scores that are obtained by a DP algorithm. Note that Navarro and Baeza-Yates[13] dealt with only edit distance.

In the first simple algorithm, we use the backward search algorithm as its subroutine to compute the ranges in the suffix array that correspond to approximately matched substrings. Given a pattern $P = P[1..m]$, our algorithm recursively enumerate the ranges of substrings (Q_i) of the text that have enough similarity to $P[1..m]$. *I.e.*, there is some string S of some size such that the alignment score of $S + Q_i$ and P has larger score than a given fixed threshold. Figure 3 shows the algorithm. Note that the alignment score of $P[1..m]$ and cQ can be easily computed in $O(m)$ time, using the final state of the dynamic programming for computing the score of P and Q. In case we use edit distance as the measure, it can be easily reduced to $O(k)$, where k is the maximum edit distance permitted.

Next, we extend the above algorithm by using a technique similar to the algorithm of Navarro and Baeza-Yates [13]. We first partition the given pattern P into h pieces for some h. Let these pieces be $R_1, R_2, \ldots, R_h$. If the alignment score of P and some sequence Q is s, then it is clear that there must be at least one piece R_i such that R_i and some substring of Q has an alignment score that is larger than or equals to s/h. Thus we can search for P as follows: First we enumerate the positions of similar substrings to these divided pieces using the divided minimum score threshold, and then check by ordinary DP algorithm the reported regions. In this way, we can avoid searching for a very long pattern that often causes large computation time when we use the first simple algorithm.

4 Experimental Results

We show the results of constructing the suffix arrays and the compressed suffix arrays for the whole DNA sequence of fly and human, and approximate matching algorithms using the indices.

4.1 Indices for the whole sequence of fly

We constructed indices for the whole DNA sequence of *fly*. The sequence has length about 98M base pairs. Therefore the size of its suffix array is about 400M bytes. On the other hand, the size of the compressed suffix array with parameters $D = 32$ and $L = 128$ is about 80M bytes. We can use three types of indices: suffix array plus text (500M bytes), compressed suffix array plus text (180M bytes), and compressed suffix array only (80M bytes). The third one is more than six times smaller than the first one.

Similarity Search based on Edit Distance We show the time for approximate matching in the sequence using the three types of indices. We first show the results of using the algorithm of Navarro and Baeza-Yates [13]. Their algorithm finds patterns from the sequence whose edit distance to a given pattern P is at most k. The edit distance between two patterns P and Q is defined as the number of operations (insertion/deletion/replacement of a character) to transform P to Q. In our experiments we set the error level of the patterns to 5%, that is, $k = 0.05m$ where m is the length of a given pattern.

Table 1: Time for approximate matching in seconds. m is the pattern length and k is the edit distance.

m	k	SA+text	CSA+text	CSA
60	3	0.0021	0.05756	0.06736
125	6	0.0039	0.10496	0.1081
250	12	0.00863	0.19593	0.20346
500	25	0.0311	0.3956	0.46036
1000	50	0.1193	0.83893	0.9439
2000	100	0.6639	2.0995	2.45596
4000	200	3.466	6.41993	7.4199

We used a Sun Ultra 450 workstation (400MHz CPU, 4G bytes memory). Table 1 shows the time for approximate matching with error level 5%. It is also depicted in Fig. 4. In the table SA+text means the search time using the suffix array plus text, CSA+text means that using the compressed

```
dpsearch(l, r): // [l, r] is the range of a pattern Q
  for each c ∈ {a, c, g, t}
    (l', r') := bsearch(l, r, c); //compute the range for a pattern cQ
    if l' ≤ r' // if cQ exists
      compute the alignment score s between P and cQ
      if s is too small to achieve the given minimum score, return.
      if s is greater than the threshold, insert [l', r'] into the range list,
           (and output corresponding regions of the text, if necessary)
        otherwise call dpsearch(l', r')
```

Figure 3: Approximate Matching Algorithm

suffix array plus text, CSA means that using the compressed suffix array only, m represents the length of the pattern P we searched for and k is the edit distance. We choose P from the whole sequence and search for its similar patterns from the whole sequence. We choose three hundred different patterns P and the execution time in the table is the average time for one execution.

We found that the search time using the compressed suffix array only is similar to that using the compressed suffix array plus the sequence. The difference is only about 10%. Furthermore, the difference in search time between using the suffix array plus text and compressed suffix array only is reasonable. The search time using the compressed suffix array is about twenty times slower than that using the suffix array for shorter patterns, and only about two times slower for longer patterns. The reason will be as follows. The approximate matching algorithm of Navarro and Baeza-Yates consists of two steps. In the first step it enumerates candidates of approximate matching patterns and in the second step the candidates are verified by dynamic programming. The suffix array or the compressed suffix array is used only in the first step and the second step takes a lot of time for longer patterns. Therefore the overhead of using the compressed suffix array becomes relatively small for longer patterns.

xternal

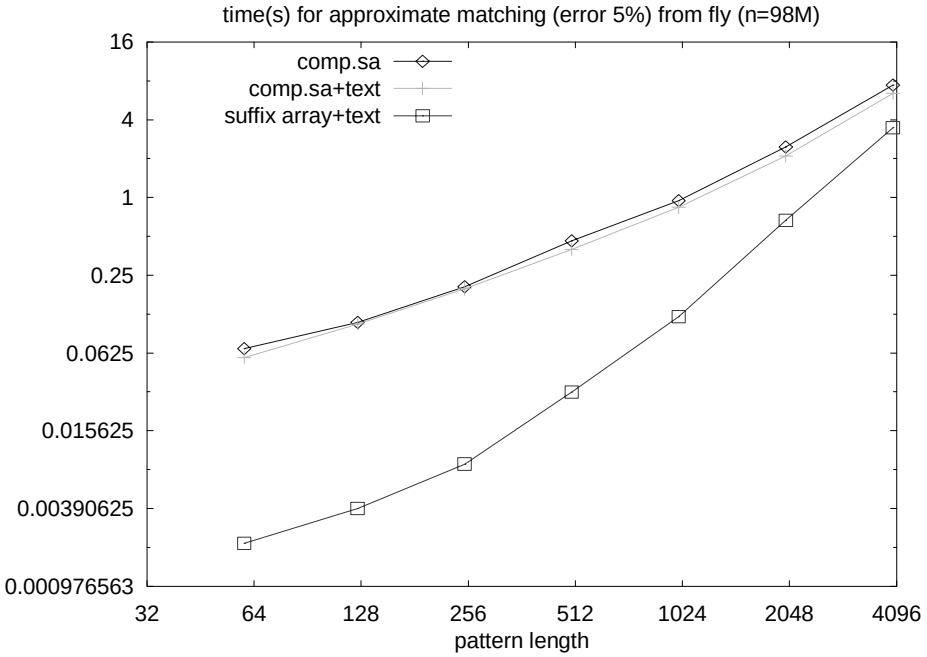

Figure 4: Time for approximate matching in seconds.

Similarity Search based on Score Matrix Next we compared our approximate string matching algorithm based on a score matrix with a naive dynamic programming algorithm. We used an IBM PC workstation with 1.7GHz CPU and 1GB memory. The score matrix we used is PAM47, in which the matching score between the same nucleotide is 5 and the mismatching score between different nucleotide is -4, and the gap penalty is -7.

Figure 5 and Fig. 6 show the time for approximate matching from the complete genome of fly using the naive DP algorithm and our algorithm using CSA respectively. We measured the time to enumerate the positions of substring which have enough similarity to a query pattern. We took

query patterns of length 20, 40 and 60 from the whole sequence of fly. The time is the average of ten executions. We used two error levels: 90% and 80%. That is, we found substrings of the sequence that have alignment scores higher than the error level times the alignment score for exact match.

We confirmed that the naive DP algorithm runs in $O(mn)$ time where m is the length of a pattern P we want to search for, and n is the length of the sequence that may contain similar patterns to P. In our algorithm, we divided query patterns into two pieces if $m = 20$, into four pieces if $m = 40$, and into five pieces if $m = 60$. Then the search time barely depends on m although it is propotional to n. Our algorithm using CSA is more than five time faster than the naive DP algorithm if error level is 80%, and about 100 times faster if error level is 90%. Concerning the size of the indices, the compressed suffix array of the whole genome of fly occupies 71057424 bytes. Because $n = 98406373$, the index occupies 5.777 bpb (bits per base). It is about 2.9 times larger than the sequence itself because the sequence occupies 2 bpb. However the compressed suffix array is about five times smaller than the compressed suffix array (27 bpb) plus the sequence (2 bpb).

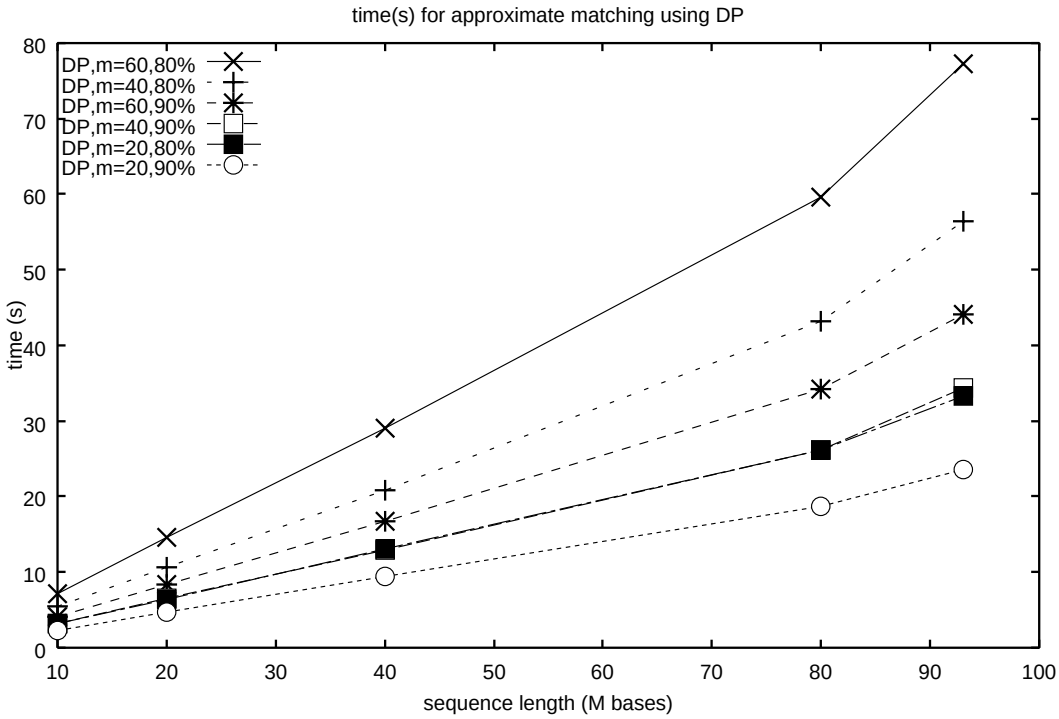

Figure 5: Time for approximate matching using DP.

4.2 Indices for the whole human genome

We have also constructed the compressed suffix array for the whole human genome. We used the April 2001 draft assembly by Human Genome Project at UCSC[1]. The size of the sequence is about 2.7 billion. Therefore its suffix array has size about 11G bytes. To construct its compressed suffix array, we used an IBM SP-2 (450MHz CPU) with 64G bytes memory. It took about 7 hours.

The Ψ function of the compressed suffix array occupies about 1060M bytes. Therefore about 3.1 bits are necessary for one character in average. For other data structures like I, J, etc., $\frac{2w}{L} + \frac{2w}{D}$ bits are necessary for one character where w is $\log_2 n$. For human genome $w = 32$ because $2^{31} < n < 2^{32}$. If $D = 32$ and $L = 128$ the size is 2.5 bits per character. Therefore the size of the compressed suffix

[1] `http://genome.ucsc.edu/`

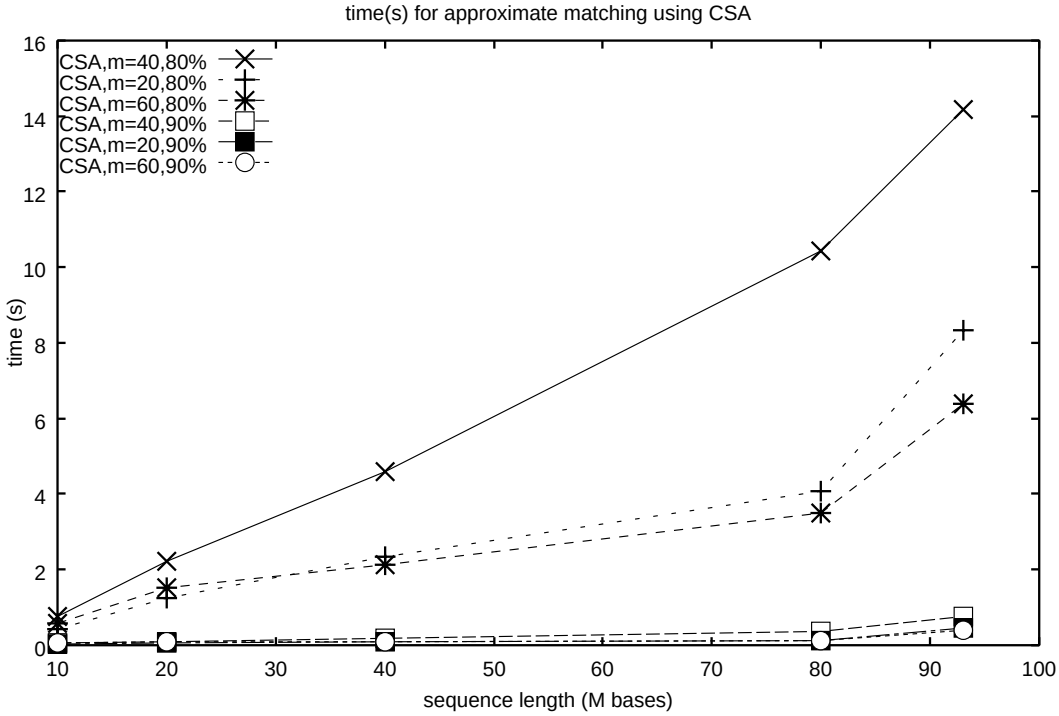

Figure 6: Time for approximate matching using CSA.

array ($D = 32, L = 128$) is about $5.6n$ bits. Because n is about 2.7 billion, the size is slightly smaller than 2G bytes. Therefore we can use it in an ordinary workstation.

5 Concluding Remarks

We have shown that the compressed suffix array has much smaller size than the suffix array in practice. We have also shown that the access time for the compressed suffix array is reasonable. The compressed suffix array can be used for various problems in genome informatics. As an application of it, we proposed algorithms for approximate string matching. Furthremore, we have constructed the compressed suffix array of the whole human genome. Its size is less than 2G bytes. Threfore it fits in main memory of an ordinary computer. This index will be used from now on.

Acknowledgement

The authors would like to thank Prof. Navarro, who kindly supplied the source codes of his approximate matching algorithm. The work of the first author was supported in part by the Grant-in-Aid for Scientific Research on Priority Areas (C), 'Genome Information Science,' from the Ministry of Education, Science, Sports and Culture of Japan.

References

[1] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM*, 41:634–647, 1994.

[2] D. R. Clark and J. I. Munro. Efficient Suffix Trees on Secondary Storage. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391, 1996.

[3] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research*, 27:2369–2376, 1999.

[4] P. Ferragina and R. Grossi. The String B-Tree: a New Data Structure for String Search in External Memory and its Applications. *Journal of the ACM*, 46(2):236–280, 1999.

[5] P. Ferragina and G. Manzini. Opportunistic Data Structures with Applications. In *41st IEEE Symp. on Foundations of Computer Science*, pages 390–398, 2000.

[6] J. Gracy and P. Argos. Automated protein sequence database classification. I. Integration of compositional similarity search, local similarity search, and multiple sequence alignment. *Bioinformatics*, 14(2):164–173, 1998.

[7] R. Grossi, K. Sadakane, and J. S. Vitter. Practical Compressed Suffix Array in Sublinear Space for Full Text Searching. working draft.

[8] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000. `http://www.cs.duke.edu/~jsv/Papers/catalog/node68.html`.

[9] E. Hunt, M. P. Atkinson, and R. W. Irving. A Database Index to Large Biological Sequences. In *To appear in Proc. VLDB 2001*, 2001.

[10] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, 29(13):1149–1171, 1999.

[11] U. Manber and G. Myers. Suffix arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[12] L. Marsan and M. F. Sagot. Extracting structured motifs using a suffix tree – algorithms and application to promoter consensus identification. In *Proc. RECOMB2000*, pages 210–219, 2000.

[13] G. Navarro and R. Baeza-Yates. A New Indexing Method for Approximate String Matching. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pages 163–185, 1999.

[14] K. Sadakane. Compressed Text Databases with Efficient Query Algorithms based on the Compressed Suffix Array. In *Proceedings of ISAAC'00*, number 1969 in LNCS, pages 410–421, 2000.

[15] P. Weiner. Linear Pattern Matching Algorihms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.