September 9, 2003

RT0548

Computer Science 8 pages

Research Report

Mining Frequent Substring Patterns with Ternary Partitioning

Yuta Tsuboi

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Mining Frequent Substring Patterns with Ternary Partitioning

Yuta Tsuboi
Tokyo Research Laboratory, IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa, Japan
yutat@jp.ibm.com

ABSTRACT

The frequent substring pattern mining problem is the problem of enumerating all substrings appearing more frequently than some threshold in a given string. This paper introduces a novel mining algorithm that is faster and requires less working memory than existing algorithms. Moreover, the algorithm generates a data structure that is useful for browsing frequent substring patterns and their contexts in the original text.
My proposed algorithm is divide-and-conquer approach that decomposes the mining task into a set of smaller tasks by using a ternary partitioning technique. After the process of frequent substring mining, the algorithm generates an incomplete suffix array which represents a pruned suffix trie based on the number of occurrences. This incomplete suffix represents the appearance positions of frequent substring patterns compactly so that their contexts can be browsed quickly. Although the average time complexity of this algorithm is $O(n \log n)$, a trial performance study shows the proposed algorithm runs faster than the pattern enumeration using a suffix tree.

Keywords

Substring Pattern Mining, Ternary Partitioning, Suffix Array, Suffix Tree

1. INTRODUCTION

The *frequent substring pattern mining problem* is the problem of enumerating all frequent substrings as patterns found in a given string.

Strings are one of the fundamental data structures and appear in many applications. Natural language sentences, DNA sequences, and time series of nominal data (e.g. Web access patterns, customer purchase behavior, etc.) are treated as strings. Nagao and Mori [14] argued that it is useful to enumerate and count all substrings in natural language data to find unknown words, compound words, and collocations. However, it is practically enough useful when analyzing such string data to enumerate only *frequent* substrings. For example, Ravichandran [17] employed frequent substrings as seeds for template in matching in a question answering system.

The *sequential pattern mining problem* [1] is a problem closely related to the frequent substring pattern mining problem. The sequential pattern mining problem is to find all frequent subsequences which include non-consecutive sequences. Although, the sequential pattern mining problem is more general than the frequent substring pattern mining problem, the sequential pattern mining problem does not differentiate continuous patterns, that is substrings, from non-contiguous patterns. Therefore, the algorithms for sequential pattern mining [1, 9] are not appropriate when only enumerating all continuous patterns. This work focuses on mining all frequent **continuous** patterns.

The frequent substring pattern mining problem might seem to be trivial, since all substrings in an input string can be traversed in $O(n)$ time (n is the length of the input string) using a well known data structure, the suffix tree [13]. In addition, a linear-time on-line algorithm which generates the suffix tree is known [20]. Therefore, all the frequent substring patterns can theoretically be found in linear time.

However, the generation algorithm for the suffix tree is somewhat complicated, so that it takes more time than simpler naive algorithms. In addition, this approach is not scalable for a large amount of input. Although, this linear time algorithm might run faster for a large amount of data, the generation algorithm uses $\Theta(n|\Sigma|)$ of working space where $|\Sigma|$ is the size of the alphabet. Thus, it is not practical to use the suffix trees for general input strings to enumerate all frequent substrings.

In this paper, a practical algorithm is introduced for the frequent substring pattern mining problem. The algorithm is practical for both scalability and pattern browsing.

The scalability is due to the following two properties:

1. small memory requirement (n computer words)
2. simple enough to run fast

The utility for pattern browsing comes from the following two properties:

1. compact representation of pattern positions
2. allows binary searches for patterns because of its lexicographically ordered enumeration

The importance of context browsing is because, in general, only a few of the patterns are useful in frequent pattern mining. Although the number of patterns is usually narrowed down using statistical measures or some heuristics during the process of mining or after the process of mining, the utility of the patterns must ultimately be assessed using human judgment. Therefore, it is necessary to browse the contextual information and search for other related patterns among the patterns to verify their usefulness.

The remainder of the paper is organized as follows. In Section 2, I define the frequent substring pattern mining problem. In Section 3, I illustrate how to enumerate frequent substrings using the suffix trees. The overview of the proposed algorithm is shown in Section 4 and the details are developed in Section 5. In Section 6, I explain the context browsing and pattern search method. The experimental and performance results are presented in Section 7. In Section 8, related work is mentioned. This work is summarized in Section 9.

2. PROBLEM STATEMENT

First, *string*, *substring*, and other key terms are defined as follows in this paper.

Definition(String) A *string* s is an ordered list of characters written consecutively. The set of the characters is denoted as Σ. In particular, \$ is used for the termination character where $\$ \notin \Sigma$ to ensure that no string is a prefix of another. For any string s , $s[i..j]$ is the *substring* of s that begins at position i and ends at position j of s. In particular, $s[i..n]$ is the *suffix* of string s that begins at position i where n denotes the length of string s.

For example, *kuras* is the substring $s[3..7]$ of the string s =*sakurasaku\$*, and *kurasaku* is the suffix $s[3..10]$ of s. In addition, a larger substring sub_i that includes the substring sub_j is called a *supersubstring pattern* of sub_j.

Next, the *frequent substring pattern mining* problem is stated as follows:

Problem Statement Let $count(p)$ be the number of occurrences of a substring p in a string s. Given a positive ξ as the *minimum support* threshold, a substring p is a *frequent substring pattern* of s if $count(p) \geq \xi$. The *frequent substring pattern mining problem* is the enumeration of all frequent substring patterns in s.

For instance, if the string s = *sakurasaku\$* and the threshold $\xi = 2$ then all frequent substring patterns are *a*, *ak*, *aku*, *k*, *ku*, *s*, *sa*, *sak*, *saku*, and *u*.

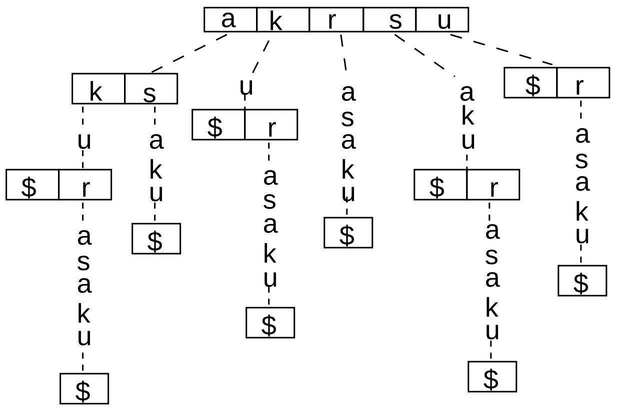


Figure 1: A Example of the Suffix Trees

3. SUFFIX TREES

In this section, I briefly introduce the suffix tree [13, 15] and illustrate how to enumerate frequent substrings using the suffix trees.

A suffix tree is a trie-like data structure that stores all suffixes of a string s. The difference between the suffix tree and the suffix trie is *path compression*, where the suffix tree eliminates nodes that have only a single child from the suffix trie [3]. In the worst case, a suffix tree can be built with a maximum of $2n$ nodes, though the suffix tree is a more space efficient data structure than the suffix trie.

Figure 1 shows the suffix tree of the example string s = *sakurasaku\$*.

In the suffix tree, a path from the root to a node represents a substring p. For the frequent substring enumeration, it is necessary to count the number of the leaf nodes that are descended from the node v. The number of the descendent leaf nodes, $count(v)$, is recursively calculated by the following equation.

$$count(v) = \begin{cases} 1 & \text{if } children(v) = \emptyset \\ \sum_{c \in children(v)} count(c) & \text{otherwise} \end{cases}$$

where $children(v)$ denotes the child nodes of the node v. Thus, it takes $O(n)$ time to calculate $count(v)$ for all nodes.

While calculating $count(e)$, the frequent **substrings** can be enumerated when $count(e) \geq \xi$. Note that an edge represents compressed nodes that have a single descendant and the path from the root to the edge represents all the substrings which appear the same number of its end node in s. Therefore, a node with $count(e) \geq \xi$ may represent multiple frequent substrings.

In practice, multiple links at nodes cause a problem for the implementations of suffix trees. The space requirements are influenced by $|\Sigma|$. Since it is impractical to use an array of size $\Theta(|\Sigma|)$ at each node for a large alphabet, it is common to use the alternative data structures (e.g. linked list, or balanced tree) to balance the constraints of space against the need for speed [7].

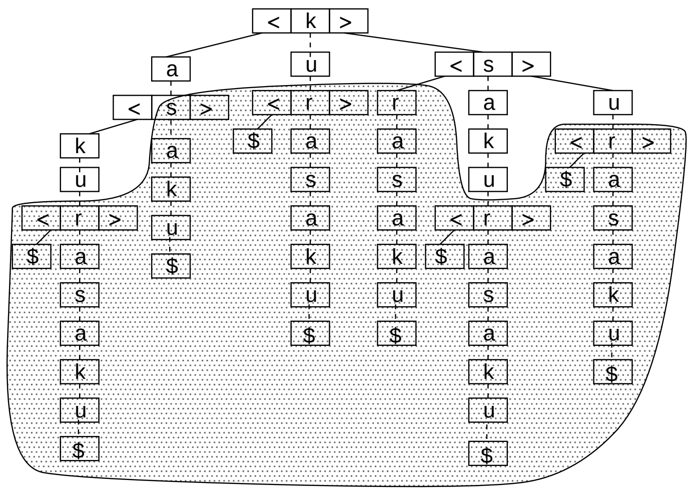


Figure 2: Pruned Ternary Search Tree

Therefore, the implementation issues have barred widespread use.

4. ALGORITHM OVERVIEW

In the previous section, how to enumerate frequent substring patterns using the suffix trees was illustrated, along with the versatility limitations of the suffix trees due to their excessive demands for working memory space. In this section and the next, I introduce and elaborate on a practical algorithm that outperforms the approach using the suffix trees.

The main idea of the proposed algorithm is based on a divide-and-conquer approach. It recursively decomposes the mining task into a set of smaller tasks using a ternary partitioning technique [5]. Let $\boldsymbol{S} = [\text{suffix}_1, \cdots, \text{suffix}_n]$ be an array of all suffixes $(s[1..n], s[2..n], \cdots, s[n-1..n], s[n])$ in a string s. The array $\boldsymbol{S}$ is divided based on the dth character of the suffixes into smaller arrays. In other word, given a partition value v, $\boldsymbol{S}$ is divided into 3 smaller arrays, $\boldsymbol{S}_=, \boldsymbol{S}_<, and \boldsymbol{S}_>$ where $\boldsymbol{S}_=$ includes suffixes whose dth character equals v, $\boldsymbol{S}_<$ includes suffixes whose dth character is lexicographically smaller than v, and $\boldsymbol{S}_>$ includes suffixes whose dth character is larger than v. If the size of $\boldsymbol{S}$, $n_=$, is greater than or equal to a minimum support threshold ξ (i.e. $n_= \geq \xi$), then it is a frequent substring pattern that begins at position 1 and ends at position d of $\text{suffix}_i \in \boldsymbol{S}_=$.

This process is recursively invoked for $\boldsymbol{S}_<$ and $\boldsymbol{S}_>$ based on the dth character and for $\boldsymbol{S}_=$ based on the $d+1$th character. The recursion stops when the size of the partitioned arrays $(n_<, n_=, n_>)$ is smaller than ξ. This reduces the search space based on the fact that any supersubstring pattern of a non-frequent pattern cannot be frequent. In addition, the recursion of an equal segment will stop if v is the terminal symbol \$ because the substring patterns can not be extended over the terminal symbol.

Here is the pseudo-code of this recursive algorithm.

The input parameters are described as follows:

$\boldsymbol{S}$ contains the suffixes of s whose $1..d-1$th characters are identical.

m is the number of the suffixes in $\boldsymbol{S}$

d is the character position of the suffixes that will be compared.

ξ is the minimum support threshold

The following function is invoked as $\mathbf{mine}(\boldsymbol{S}, n, 1)$:

```
mine($\boldsymbol{S}, m, d$)
  if $m < \xi$ then
    return
  end
  select a partitioning value $v$
  partition $\boldsymbol{S}$
    by comparing $\text{suffix}_i[d]$ ($\text{suffix}_i \in \boldsymbol{S}$) with $v$
    to form $\boldsymbol{S}_<, \boldsymbol{S}_=, \boldsymbol{S}_>$
    (their sizes are $n_<, n_=, n_>$)
  mine($\boldsymbol{S}_<, n_<, d$)
  if $n_= \geq \xi$ and $v \neq \$$ then
    print $\text{suffix}_i[1..d]$($\text{suffix}_i \in \boldsymbol{S}_=$) as pattern
    mine($\boldsymbol{S}_=, n_=, d+1$)
  end
  mine($\boldsymbol{S}_>, n_>, d$)
```

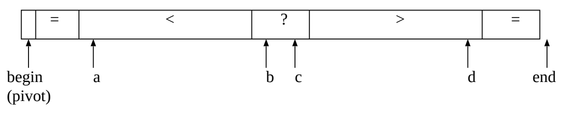
This mining process is identical to walking through a *pruned ternary search tree* are stored in which all of the suffixes of s. The nodes of this ternary search tree are ignored where the number of descendent leaf nodes is smaller than ξ. Figure 2 shows the pruned ternary search tree for the example string $s = sakurasaku\$$.

The algorithm is similar to the *multikey quicksort* proposed by Bentley and Sedgewick [6] except for the pruning phase. The multikey quicksort is an effective sorting algorithm for a set of vectors, such as strings. In a similar way, the proposed algorithm partially sorts the vectors while ignoring the terminal elements of vectors that are not frequent. Just as for the multikey quicksort, the expected time complexity of the proposed algorithm is $O(n \log n)$ if the partition value is chosen as the median of all of the dth character of suffixes $\boldsymbol{S}$.

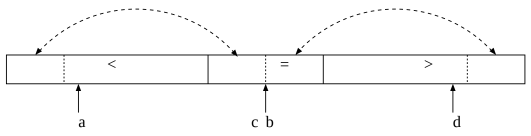
5. IMPLEMENTATION OF THE ALGORITHM

Although the array of suffixes $\boldsymbol{S}$ is used for the overview of the algorithm in Section 4, this naive approach is inefficient since the size of $\boldsymbol{S}$ is $\Theta(n(n-1)/2)$. In this section, I present a space efficient implementation using the indexes that represent each suffixes of the string.

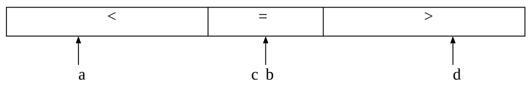
The key idea in the following implementation is that the frequent substrings are enumerated during the process of building the *incomplete suffix array.* The solution involves using suffice arrays to retain that substring information. A suffix array [12, 7] is an array of the integer indexes in the range 1 to n, specifying the lexicographic order of the n suffixes of string s , and this is a very space efficient representation of a *suffix trie.* The suffix array requires only n computer words for an input string of the length, n. The proposed algorithm builds an incomplete suffix array because of the partial sorting properties which were mentioned at the last section.



(a) starts partitioning from both sides



(b) exchanges the equal regions



(c) the end of ternary partitioning

Figure 3: Ternary Partitioning

Since the outline of the algorithm was already described the above, the remainder of this section shows the proposed algorithm implementation as concretely applied to the strings of C++ programs. Note that the following sample code follows the same style as the *Program 1. A C program to sort strings* which appears in the paper describing multikey quicksort [6].

The reader is assumed to be familiar with the C++ programming language [19]. In the following sample code, an input string *str* is of **string** type and the suffix array *idx* is a **vector** of the **size** type of the **string**. Both are freely accessible from the main function as global variables or member variables. The input strings are assumed to end with the terminal symbol $ and the constant value *EOS* is a **value** type of **string** and is used to represent $ in this sample code. The minimum support threshold *minsup* is the **size** type of *idx* and is also freely accessible, like the *str* and the *idx*. Note that it is assumed that *minsup* > 2 in this implementation.

```
typedef string::size_type s_t;
typedef string::value_type v_t;
typedef vector<s_t>::size_type i_t;
string str;
vector<s_t> idx;
i_t minsup;
```

The initial element values of the suffix array *idx* are the positions of the input *str* as follows:

```
for(s_t i = 0; i < str.size(); i++) {
  idx.push_back(i);
}
```

Here is the complete mining algorithm:

```
void mine(i_t begin, i_t end, s_t depth, bool equal=false)
{
    i_t count = end - begin;
    if(count < minsup) {
        return;
    } else if (equal) {
        printPattern(begin, count, depth);
    }

    i_t pivot = selectPivot(begin, end);
    swap(idx[begin], idx[pivot]);
    v_t t = getValue(begin, depth);
    i_t a = begin+1, c = end-1;
    i_t b = a      , d = c;
    v_t r;
    while(true) {
        while(b <= c && ((r=getValue(b, depth)-t) <= 0)) {
            if (r == 0) { swap(idx[a], idx[b]); a++; }
            b++;
        }
        while(b <= c && ((r=getValue(c, depth)-t) >= 0)) {
            if (r == 0) { swap(idx[c], idx[d]); d--; }
            c--;
        }
        if(b > c) {
            break;
        }
        swap(idx[b], idx[c]);
        b++;
        c--;
    }
    i_t range = min(a - begin, b - a);
    vectorSwap(begin, b - range, range);
    range = min(d - c, end - d - 1);
    vectorSwap(b, end - range, range);

    range = b - a;
    mine(begin, begin + range, depth);

    if(t != EOS) {
        mine(begin+range, range+a+end-d-1, depth+1, true);
    }
    range = d - c;
    mine(end - range, end, depth);
}
```

The above main function is initially called as:

```
mine(0, str.size(), 0);
```

The followings are the auxiliary functions of the main mining function:

The *getValue* function returns the *depth*th character of the suffix *i*:

```
v_t getValue(i_t i, s_t depth)
{
    return str[idx[i] + depth];
}
```

The *vectorSwap* function moves sequences of equal elements from their temporary positions on both sides of the *idx*. Because this vector swap is a kind of tricky, Figure 3 illustrates this process.

```
void vectorSwap(i_t i, i_t j, i_t len)
{
    while(len-- > 0) {
        swap(idx[i], idx[j]);
        i++;
        j++;
    }
}
```

Although the partitioning value can be selected in many ways, this *selectPivot* function returns the index of the partitioning value at random.

```
i_t selectPivot(i_t begin, i_t end)
{
    return begin + rand() % (end - begin);
}
```

The *printPattern* function outputs the frequency of a substring pattern and the pattern itself to the standard output stream.

```
void printPattern(i_t offset,i_t count,v_t depth)
{
    cout << count << "\t"
         << str.substr(idx[offset], depth)
         << endl;
}
```

6. PATTERN BROWSING

Usually, the outputs of frequent pattern mining algorithms are only the patterns found and their degree of support. However, not all the patterns are interesting. The valuable patterns still have to be selected by humans. Thus, it is necessary to browse the patterns and their contextual information for the patterns to verify their significance. In particular, for natural language data, the contextual information helps in the disambiguation of word senses.

From that perspective, the proposed algorithm has a good utility because the incomplete suffix array provides a compact representation of the pattern positions, and the lexicographically ordered enumeration of patterns allows for binary searches among them.

To describe pattern positions, the following three positive integers should be output: (1) the length of the substring pattern *depth*, (2) the *offset* of the suffix array *idx*, and (3) the *count* of the appearances of the pattern. These three integers plus the incomplete suffix array are sufficient to describe all of the appearance positions of the pattern and the string representation of the pattern.

The string representation of the pattern can be described as $s[idx[offset]..idx[offset]+depth]$. All positions of the patterns are described as $idx[offset], \cdots, idx[offset+count]$, so that the context of patterns can be quickly collected using that positional information and the original string.

	parameter	fixed value	data type
1	minsup(0.0001% – 0.001%)	data size(5 and 4.4MB)	papers, DNA
2	data size(1 – 20MB)	minsup(0.001%)	papers

Table 1: Experimental Parameters

7. EXPERIMENTS

7.1 Experimental Methodology

Table 1 summarizes these two experiments.

To show the effectiveness of the approach, I conducted two experiments to compare the enumeration using the suffix tree and the proposed algorithm.

The Ukkonen algorithm [20] was employed to build the suffix trees. As I mentioned in Section 3, the space requirements of the suffix trees depends on the data structure used for multiple links at each node. In these experiments, both balanced trees [1] and sorted vectors [2] were used. Given k is the number of children at a node, the implementation using the balanced trees requires $O(\log k)$ time for additions and searches among its children. The suffix tree node using sorted vector requires $O(k)$ time for additions and $O(\log k)$ time for searches.

The implementation choice of the presented algorithm (ternary-partitioning) is how to select the partitioning value. In these experiments, I employed two selection methods, one selecting a random value as used in the sample code (*rand*), in the *selectPivot* function, and the other selecting the median of the sampled points as a pseudo-median (*pseudomedian*).

Both algorithms were implemented in C++ programming language and compiled by gcc (version 3.2) with optimization flag -O3, and executed on Windows2000 (Pentium4 1.5GHz and 1.5GB main memory).

Two types of input data were employed, a natural language text and a DNA sequence. For the natural language text data, English papers in the Computer Science domain were concatenated as one long string. Those papers were downloaded from the e-Print archive [3]. The size of the alphabet $|\Sigma|$ was 89 which included control characters and the total amount of data was 20MB. For the DNA sequence, the unannotated sequence of E. coli, strain K-12, substrain MG1655, version M52 was used, as available from the genome center at the University of Wisconsin [4]. The alphabet $|\Sigma|$ size was 4 (AGTC) and the data size was 4.4MB.

In the first experiment, the scalabilities of both approaches were compared as the minimum support threshold ratios decreased from 0.001% to 0.0001% on both the English papers and the DNA sequence. The minimum support threshold ratio is derived from the ratio between the minimum support count and the data size. For example, the minimum support count decreases from 52 to 5 for 5MB of string data.

In the second experiment, the scalabilities of both approaches

[1]map in C++ Standard Template Library
[2]the AssocVector developed by Alexandrescu[2]
[3]`<http://www.arxiv.org/>`
[4]`<http://www.genome.wisc.edu/sequencing/k12.htm>`

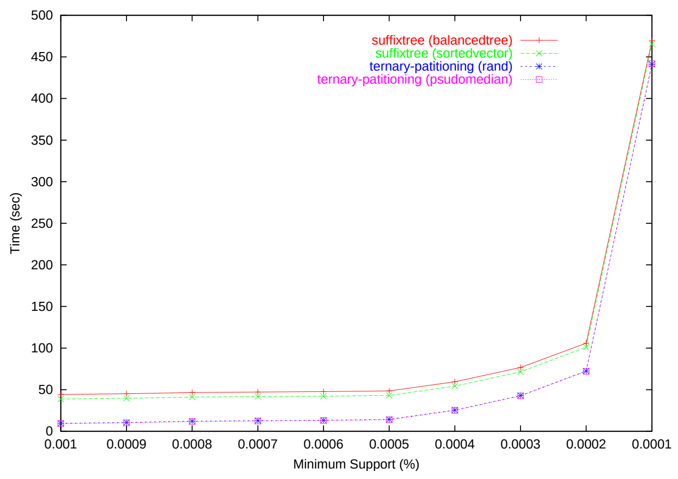


Figure 4: Execution Time with threshold, e-print paper (5MB)

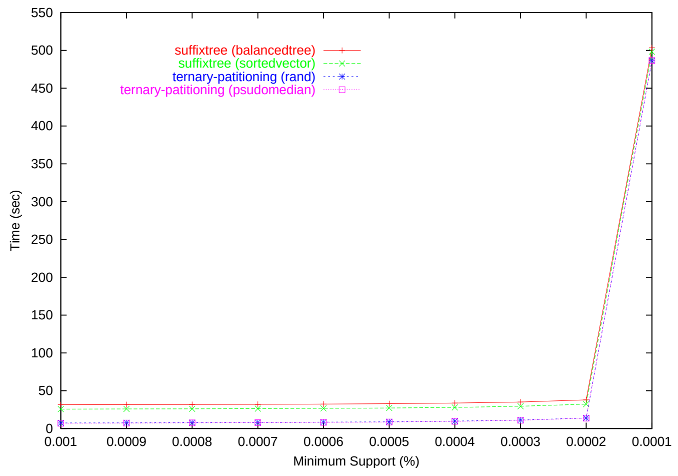


Figure 5: Execution Time with threshold, E. coli genome sequence (4.4MB)

were compared as the data size increased from 1MB to 20MB for the English papers. The minimum support ratio was fixed at 0.001% in the second experiment.

7.2 Results and Discussion

The empirical performance is depicted in Figure 4, 5 for the first experiment and in Figure 6 for the second experiment. These execution times are the averages of three trial and do not include the reading time for the input strings from disk storages.

On the scalability tests with varying thresholds, the proposed approach (*ternary-partitioning*) outperformed the suffix tree approach (*suffixtree*) on both natural language text (Figure 4) and DNA sequence data (Figure 5). On average, the proposed approach is 28.7 seconds faster for the text and 17.8 seconds faster for the DNA data than the suffix tree approach. The reason why the performances of both approachs are close may be that the output cost of substring patterns is relatively higher at the low minimum support levels. For example, the number of the frequent patterns over the threshold is 604K with a threshold of 0.0003%, 989K with 0.0002%, and 3.8 M with 0.0001%. The amount of memory used for each algorithm is described in Table 2.

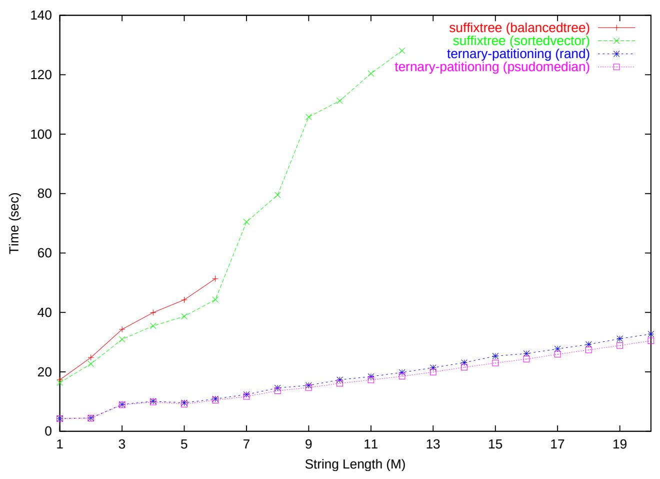


Figure 6: Execution Time with String Length

algorithm	memory usage (MB)
suffixtree(balancedtree)	693
suffixtree(sortedvector)	391
ternary-partitioning(both)	26

Table 2: Memory Usage for 5MB of text

On the scalability tests with varying data sizes, the proposed approach scaled much better than the suffix tree approach (Figure 6). On average, the new approach is five times faster than the suffix tree approach. Note that the implementations of the suffix tree have cause swapping at more than 6 MB of data (balanced tree) or 12 MB of data (sorted vector).

The selection methods in the proposed algorithm or the data structure used in suffix tree implementations caused little performance difference. For the suffix tree approach, the *sortedvector* approach is faster and more space efficient than the *balancedtree* approach. This result is because the cost of balancing is somehow high in the implementation of *balancedtree* in C++ [4]. For the proposed approach, the *pseudomedian* outperformed the *rand*.

Overall, the presented ternary-partitioning algorithm is much more scalable than the algorithm using suffix trees. This relationship between the suffix tree approach and the proposed approach is similar to the relationship between linear-time sorting algorithms and quicksort. According to Sedgewick [18], although the linear-time algorithms, like the radix sort, are sometimes good for special applications, they are not as good as quicksort for general use. In the same way, the suffix tree is not as fast as expected.

8. RELATED WORK

Krishnan et al. [16] introduced a *count-suffix tree*, which stores a count for each substring in the tree instead of stor-

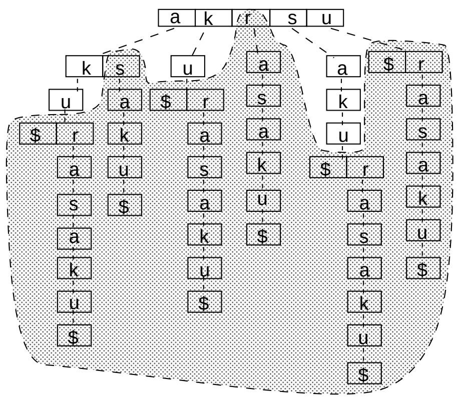


Figure 7: Pruned Suffix Trie

ing the pointers to the occurrences of the substring in the original string. They used the *pruned count-suffix tree* for the selectivity estimation of the substring query. Although the frequent pattern enumeration may be sped up by using this pruned count-suffix tree, either the on-line or the linear time properties of the Ukkonen's algorithm [20] must be abandoned to build the count-suffix tree.

Kasai et al. [10] proposed a linear time algorithm that traverses all substrings using a suffix array. Their algorithm runs in linear time with additional information stored in a data structure, called the height array. However, it needs the suffix array to be prepared in advance, so that this approach requires much more computational cost than our approach.

Nagao and Mori [14] proposed a data structure that enumerates all substrings and argued that it is useful for extracting unkown words, compound words, and collocations. Actually, their proposed data structure is equal to the suffix array and the height array [12].

The partitioning-based divide-and-conquer method has recently garnered attention in the data mining community. Although the traditional data mining algorithms adopt an Apriori-like candidate-generation-and-test approach, this partitioning based divide-and-conquer method led to new ways of thinking about data mining. Han et al. [8] proposed a novel data structure, an *FP-tree* which compresses a database and uses a divide-and-conquer method to decompose the mining task into a set of smaller tasks for mining limited patterns in conditional databases. Pei et al. [9] introduced a divide-and-conquer approach for the sequential pattern mining problem, called *PrefixSpan*. PrefixSpan narrows the search space based on the prefixes of sequential patterns in a recursive manner. It uses each frequent item to partition the sequential database into a set of smaller databases sharing the item as the prefix of the patterns to be found. Then it recursively searches for frequent subsequence patterns in each smaller database.

Kudo et al. [11] extends the PrefixSpan algorithm to enumerate frequent substring patterns. The algorithm partitions the suffix array into smaller suffix arrays, such that the number of the smaller arrays is the same as the size of alphabet. Thus, their approach is identical to searching a pruned suffix trie which stores all of the suffixes of s. Figure 7 shows an example using the pruned suffix trie of $s =$ *sakurasaku$*. The nodes are pruned if the number of descendent leaf node is smaller than $\xi = 2$.

The presented approach adopts a patitioning-based divide-and-conquer method. In particular, the PrefixSpan algorithm is closely related to my algorithm. The difference besides the different problem to be solved is that the presented approach does not create new databases or indexes at the time of partitioning. PrefixSpan and Kudo's algorithm both generate a set of smaller databases (or indexes for the *pseudo-projection procedure*). The in-place property of the ternary partitioning technique achieves more space efficiency.

9. CONCLUSIONS

This study introduced the frequent substrings pattern mining and presented a novel algorithm for solving this problem. The empirical result shows the proposed algorithm outperforms the approach using the suffix trees which might seem to be the magical tool for substring problems. In particular, the proposed algorithm scales much better than those using suffix trees. In addition, the proposed algorithm has convenient properties for browsing the frequent patterns and their contextual information.

10. ACKNOWLEDGMENTS

The author would like to thank Mr. Tetsuro Shibuya for his help in the implementation of the suffix trees and the DNA sequence data.

11. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. 11th Int. Conf. Data Engineering, ICDE*, pages 3–14. IEEE Press, 6–10 1995.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[3] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[4] J. L. Bentley. *Programming Pearls.* Addison-Wesley, second edition, 2000.

[5] J. L. Bentley and M. D. McIlroy. Engineering a sort function. In *Software–Practice and Experience (SPE)*, volume 23, pages 1249–1265, 1993.

[6] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1997.

[7] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Intl. Conference on Management of Data.* ACM Press, 2000.

[9] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. on Data Engineering (ICDE'01)*, pages 215–224, Heidelberg, Germany, April 2001.

[10] T. Kasai, H. Arimura, and S. Arikawa. Efficient substring traversal with suffix arrays. Technical report, Department of Informatics, Kyushu University, 2001.

[11] Kudo Taku, Kaoru Yamamoto, Yuta Tsuboi, Yuji Matsumoto. Text mining using linguistic information. In *IPSJ SIGNL-148 (in Japanese)*, 2002.

[12] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[13] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.

[14] M. Nagao and S. Mori. A New Method of N-gram Statistics for Large Number of n and Automatic Extraction of Words and Phrases from Large Text Data of Japanese. In *the 15th International Conference on Computational Linguistics*, 1994.

[15] M. Nelson. Fast string searching with suffix trees. *Dr. Dobb's Journal*, 1996.

[16] P. Krishnan and Jeffrey Scott Vitter and Bala Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 282–293, 1996.

[17] D. Ravichandran and E. Hovy. Learning surface text patterns for a question answering system. In *Annual Meeting of the Association for Computational Linguistics*, 2002.

[18] R. Sedgewick. *Algorithms in C++ (second edition)*. Addison-Wesley, 1992.

[19] B. Stroustrup. *The C++ Programming Language, 2d edition.* Addison-Wesley, 1991.

[20] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.