NAIST-IS-MT9951064

Master's Thesis

Writing Assistance through Search Techniques

Satoru Takabayashi

February 9, 2001

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

Master's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
MASTER of ENGINEERING

Satoru Takabayashi

Thesis Committee: Yuji Matsumoto, Professor
Katsumasa Watanabe, Professor
Yasuharu Den, Associate Professor

Writing Assistance through Search Techniques*

Satoru Takabayashi

Abstract

Traditionally, people used to write by hand. Nowadays, people seldom write by hand, instead people write with computers. However, still there are many scopes to improve the writing environment because searching process is not seamlessly integrated with writing process. By regarding writing as a process of searching for words or expressions, it can be concluded that writing is, in a broad sense, a process of searching. We take the fact seriously and propose several methods for writing assistance through search techniques. In this thesis, we realize input acceleration systems and proofreading systems through our three complete search systems: *Namazu*, *Sary*, and *Migemo*.

Keywords:

Searching, Writing Assistance, Suffix Array, Incremental Search, Input Prediction, Proofreading

*Master's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT9951064, February 9, 2001.

検索技術を用いた作文支援*

高林 哲

内容梗概

ペンを握って文章を書くという旧来のスタイルと違い、現代では計算機上で文章を書くことができる。しかし、作文と検索の過程には依然として障害があり、理想的な作文環境とはほど遠い。作文の過程を、言葉を探すという行為の連続と捉えれば、作文と検索は密接に結びついているといえる。本論文では、この関係に注目し、検索技術を用いた作文支援の手法を提案する。基盤となる検索技術として *Namazu*, *Sary*, *Migemo* の 3 つの検索システムを実装し、それらを用いて、入力および文章校正を支援するシステムを構築した。

キーワード

検索, 作文支援, 接尾辞配列, 漸進的検索, 入力予測, 文章校正

*奈良先端科学技術大学院大学 情報科学研究科 情報処理学専攻 修士論文, NAIST-IS-MT9951064, 2001 年 2 月 9 日.

Contents

1 Introduction **1**

1.1. Problems of the Current Writing Environment 2

1.2. Our Methodology: “Writing is Searching” 3

1.3. Organization of the Thesis 4

2 Search Techniques **5**

2.1. Sequential String Searching 5

2.2. Approximate String Searching 6

2.2.1 Dynamic Programming 6

2.2.2 Non-deterministic Finite Automaton 6

2.2.3 An Advanced Algorithm 7

2.3. Regular Expression Searching 8

2.4. Incremental Searching 8

2.5. Inverted File 9

2.5.1 Construction of Inverted File 9

2.5.2 Searching with Inverted File 11

2.5.3 Compression of Inverted File 11

2.6. Suffix Array 12

2.6.1 Construction of Suffix Array 12

2.6.2 Searching with Suffix Array 13

2.6.3 Applications of Suffix Array 13

2.7. Comparison between Inverted File and Suffix Array 14

3 Our Search Systems **17**

3.1. Namazu 17

3.2. Sary 19

3.2.1 Block Sorting 19

3.2.2 Merging . . . 20
3.2.3 Performance . . . 21
3.3. Migemo . . . 23
3.3.1 Dynamic Pattern Expansion . . . 24
3.3.2 Applications . . . 25

4 Input Acceleration **27**
4.1. Dynamic Abbreviation . . . 27
4.2. Quick Copy-and-Paste . . . 28
4.3. Input Prediction . . . 29

5 Proofreading **32**
5.1. Finding Sample Sentences . . . 32
5.2. Checking Expressions . . . 34

6 Conclusions and Future Work **36**

References **38**

List of Figures

1.1 Traditional and modern writing environments. 1
1.2 Relation between human and computer. 2
1.3 Three types of information and their extents. 3

2.1 Three types of errors in the pattern 6
2.2 The dynamic programming for searching the text 'failure' for the pattern 'figure'. The lower right element indicate the finding with two errors. 7
2.3 The NFA for the pattern 'figure' with two errors. The shaded states and the bold lines illustrate the successful transition of reading the text 'failure'. 7
2.4 The non-deterministic and deterministic finite automata for the pattern `a*(a|bb)c`. 8
2.5 Process of incremental searching for 'scheme'. 9
2.6 The sample text and the position of each word. 10
2.7 The inverted file for the sample text. 10
2.8 The suffixes for the sample text and their index points. 12
2.9 The suffixes sorted in lexicographical order. Each index point is reordered according to its corresponding suffix. 13
2.10 The suffix array for the sample text. Each entry corresponds to an index point shown in Figure 2.9. 13
2.11 Searching the sample text for 'than' by binary search using the suffix array. Numbered arrows show the order of the process. 14
2.12 Morphological analysis with ChaSen. 16

3.1 Relation among our systems. 17
3.2 The mailing list search engine using Namazu at the GNOME website. 18
3.3 Merging the sorted partial blocks in a binary fashion. 20
3.4 Merging the sorted partial blocks with the heap structure. 21
3.5 Suffix array construction with the Intel, Alpha, and MIPS machines. The right graphs shows the results with multi-threading. 22

3.6 Process of incremental searching for '活発' with Migemo. 24
3.7 The sequential dynamic pattern expansions for for 'k', 'ka', and 'kap'. 25
3.8 Searching a Japanese word '情報' with an English word 'information'. 26
3.9 Conceptual searching for a person's name. 26

4.1 Dynamic abbreviation for 'floccinaucinihilipilification'. 27
4.2 Dynamic abbreviation for '奈良先端科学技術大学院大学'. 28
4.3 Process of quick copy-and-paste. The user reuses information about a recently-published book. 30
4.4 Inputting 'user interface' with Pen-based POBox. 30
4.5 Process of input prediction with the previously written sentence. 31

5.1 Finding sample sentences for 'morphological'. The user only types 'morphol' at the moment. 33
5.2 Finding sample sentences for 'thanks to'. 33
5.3 Checking the expression 'Most people have difficulties in using computers' with our system. 34
5.4 Checking the expression 'this are pens' 35

List of Tables

2.1 BER compression for 32 bit positive integers. 11
2.2 Comparison between suffix array and inverted file. 14

3.1 Suffix array construction with a memory-poor machine. The block size is set to 4 MB. . . 20
3.2 Platforms used for the experiments on suffix array construction. 23

CHAPTER 1 *Introduction*

I don't think necessity is the mother of invention — invention in my opinion arises directly from the idleness, possibly also from laziness. To save oneself trouble.
— Agatha Christie

Traditionally, people used to write by hand. Nowadays, people seldom write by hand, instead people write with computers through keyboards, and the opportunity of handwriting will decrease rapidly in the future. Figure 1.1 illustrates the both writing environments side by side.

In traditional writing environment, people used to struggle with piles of dictionaries and documents to find necessary information. This work takes a long time and make people exhausted. Moreover, people had to take notes on own index cards and information sharing and reuse was poorly realized.

In contrast, modern computer-aided writing environment solves these problems to a certain extent. People can consult electronic dictionaries on computers quickly and find vast quantities of shared information on the Internet. Additionally, people can do copy-and-paste or edit sentences freely and correct spellings and even grammars semi-automatically with computers.

However, still there are many scopes to improve the writing environment. In this thesis, we propose several methods for realizing a better writing environment.

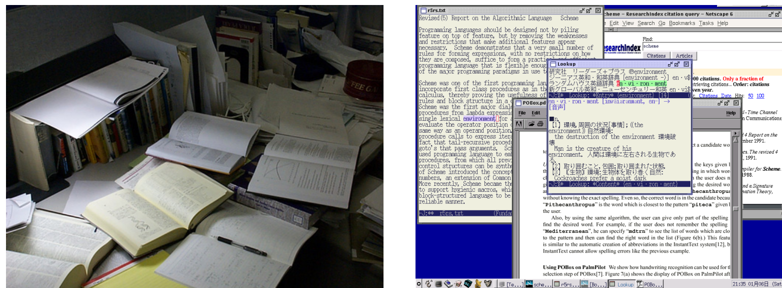


Figure 1.1. Traditional and modern writing environments.

1.1. Problems of the Current Writing Environment

Although computers blessed us with better writing environments, there are still problems. One problem is the usability of computers. Most people have difficulties in using computers, that is learning and mastering computers impose heavy burdens on people. As a result, people cannot concentrate on their ideas and ideas are restricted by their tools (i.e. computers). In other words, computers interfere with train of thoughts.

Figure 1.2 illustrates a simplified role model in computer-aided writing. First, questioning is human activity casting a wish to the computer search for something. Second, searching is mainly computer's work because of its memory capacity. Finally, writing is a collaboration between human and computer through the appropriate devices like a keyboard.

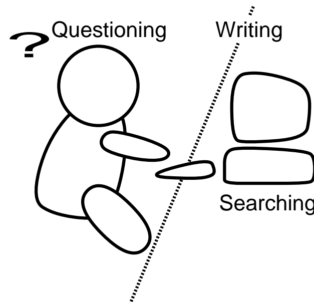


Figure 1.2. Relation between human and computer.

When people want to write something, they find words which convey their thought. Since human memory has limited capacity, much information is stored externally. In other words, human memory performs very efficiently with the help of references. For example, dictionaries, thesauri, and encyclopedias are typically well organized as external memory. Additionally, inexhaustible information is available on the Internet or somewhere. Figure 1.3 depicts three types of information and their extents.

We frequently search external and "available somewhere" information by several ways for a various kind of writing. These types of information are especially necessary for writing technical reports or exchanging information with others by machine readable means. Typical process of such kind of writing is summarized as follows:

1. Search information.

2. Copy-and-paste the information.

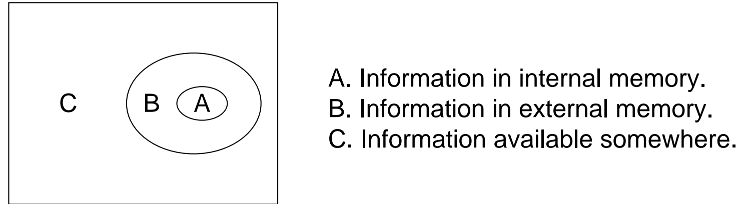


Figure 1.3. Three types of information and their extents.

3. Edit the information.

4. Compose messages with the information.

This process requires both text editing and information retrieval simultaneously. However, these two tasks have not been integrated with the current writing environment. In fact, people have to switch the tools for text editing and information retrieval by turns. For example, if you want to use information on the Internet in your writing, you have to leave from a text editor and move to a web browser for retrieving information and then go back to the text editor again to carry the retrieved information. These operations make people disoriented and exhausted. We argue that text editing and information retrieval should be seamlessly integrated because the process of writing is a seamless process of searching and composition.

Another problem of the current writing environment is the persistent needs for printed information. Although the Internet provides vast quantities of electronic information, printed information such as books is still important source of information. One reason why electronic books are not yet popular currently is that computer screens have poor visibility than papers. In fact, people often obtain information from the Internet but print them for reading. Thus, information which is not on computers should also be used for writing. Not to mention, printed information is difficult to reach and reuse. Consequently, our desks still tend to be piled with papers but this issue is beyond the scope of this thesis and we will not discuss it more.

1.2. Our Methodology: "Writing is Searching"

As discussed above, people search information frequently for writing. Moreover, by regarding writing as a process of searching for words or expressions, it can be concluded

that writing is, in a broad sense, a process of searching. We take the fact seriously and call it "Writing is Searching."

In this thesis, not only do we present several methods for writing assistance employing various search techniques, but also we propose the methodology "Writing is Searching." The integration of writing and searching opens a way for a better writing environment.

1.3. Organization of the Thesis

This thesis is organized as follows. Chapter 2 briefly describes background search techniques. We emphasize that a *suffix array* can be used for applied text processing which uses large corpora. In Chapter 3, we present the implementations of our search systems: *Namazu*, *Sary*, and *Migemo*. Input acceleration with the help of dynamic string searching is discussed in Chapter 4. Chapter 5 proposes methods for proofreading system employing large corpora as example-base. Conclusions and future work are presented in Chapter 6.

CHAPTER 2 *Search Techniques*

There are two types of knowledge. One is knowing a thing. The other is knowing where to find it.
— Samuel Johnson

Since human memory has limited capacity, searching of external information is essential task for writing. In this chapter, we discuss various search techniques with computers.

2.1. Sequential String Searching

The simplest string search algorithm is brute-force. It sequentially traverses the target text character by character and tries to match the given pattern (i.e. keyword) at all positions in the text. The problem of the algorithm is the necessity of backtracking. Let the text be 'abcefg' and the pattern be 'abcd', the algorithm tries to match the pattern from the first position of the text. It fails at the position of 'e' in the text and proceeds to match the pattern from the second position of the text (i.e. 'bcefg') and so on. In this case, the algorithm takes four comparisons just for proceeding one character in the text. Let m be the length of the pattern and n be the length of the text, the worst-case of the algorithm have a time complexity of $O(mn)$. Nonetheless, the brute-force algorithm is reasonably fast for usual cases.

Knuth-Morris-Pratt[14] and *Boyer-Moore*[7] are linear algorithms in their worst-case. The former is not so fast but theoretically valuable and the later is fast and widely used. Practical comparison of performance among these algorithms are described in Chapter 8 of [4]. Aoe[3] collects classical papers in this field.

Even though efficient algorithms are employed, searching a large text file sequentially takes too long time. However, such a large text can be searched very fast with appropriate data structures such as an inverted file and a suffix array. We will discuss these data structures in Section 2.5 and Section 2.6, respectively.

2.2. Approximate String Searching

Approximate string searching is also called *searching allowing errors*. Approximate string searching finds the text position where the pattern occurs with at most a certain number of errors. Figure 2.1 depicts three types of errors: *Replacement*, *Insertion*, and *Deletion*. Although each error can have different weight, we consider all errors with equal weight for the sake of simplification.

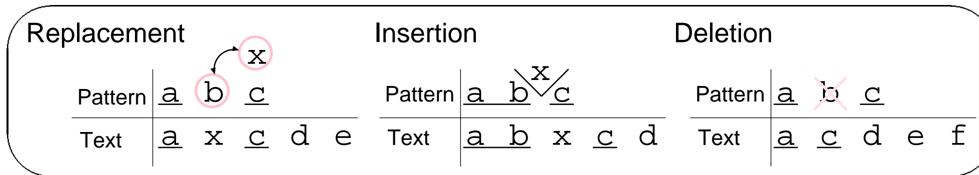


Figure 2.1. Three types of errors in the pattern

2.2.1 Dynamic Programming

First, we discuss an approach based on dynamic programming for approximate string searching. Let $t_{1..n}$ be the text and $p_{1..m}$ be the pattern, then the entry of matrix $E[i, j]$ represents the minimum number of errors permitted in for matching $p_{1..i}$ $(i \leq m)$ to $t_{1..j}$ $(j \leq n)$. The entries on the first column $E[0, j]$ $(0 \leq j \leq n)$ are initialized to 0 and the entries on the first row $E[i,0]$ $(0 \leq i \leq m)$ are initialized to i. Other entries $E[i, j]$ $(1 \leq i \leq n, 1 \leq j \leq m)$ are dynamically computed column by column as searching proceeds. The pseudo code is listed as follows:

if $p_i = t_j$ **then**

$E[i, j] := E[i-1, j-1]$

else

$E[i, j] := 1 + \min(E[i-1, j-1], E[i-1, j], E[i, j-1])$

end

Figure 2.2 shows an example of this algorithm. The cost of the algorithm is $O(mn)$ because $m \cdot n$ computations are required for completing the matrix.

2.2.2 Non-deterministic Finite Automaton

Another algorithm employs non-deterministic finite automaton (NFA). The algorithm constructs the NFA from the pattern and changes the states of the NFA as reading the

text (t_j)

pattern (p_i)		f	a	i	l	u	r	e
	0	0	0	0	0	0	0	0
f	1	0	1	1	1	1	1	1
i	2	1	1	1	2	2	2	2
g	3	2	2	2	2	3	3	3
u	4	3	3	3	3	2	3	4
r	5	4	4	4	4	3	2	3
e	6	5	5	5	5	4	3	2

Figure 2.2. The dynamic programming for searching the text 'failure' for the pattern 'figure'. The lower right element indicate the finding with two errors.

text character by character. Figure 2.3 illustrates an NFA allowing two errors. The NFA accepts the text position of the pattern with at most two errors occurs as the end of matching if one of the rightmost states is active. Non-labeled transitions consume errors in the pattern.

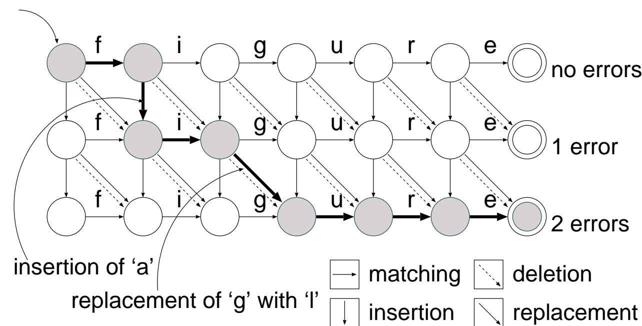


Figure 2.3. The NFA for the pattern 'figure' with two errors. The shaded states and the bold lines illustrate the successful transition of reading the text 'failure'.

2.2.3 An Advanced Algorithm

Wu and Manber[29] proposed an advanced algorithm based on *bit-parallelism* for fast approximate string search, which also accommodates matching regular expressions. Bit-parallelism reduces the computations for pattern matching by taking advantage of the fast bit operations of CPUs. The Unix command, *agrep*[28] is the their implementation of their proposed algorithm.

Chapter 8 of [4] briefly summarizes this field and Hall[10] surveys this field compre-

hensively including a probabilistic similarity approach.

2.3. Regular Expression Searching

Regular expression searching finds the text position at which the regular expression pattern matches. The process constructs an automaton from the pattern and then changes the states of the automaton as reading the text character by character. Figure 2.4 illustrates the non-deterministic automaton (NFA) and the deterministic automaton (DFA) for the pattern `a*(a|bb)c`. The UNIX command *grep* is a popular tool for regular expression searching.

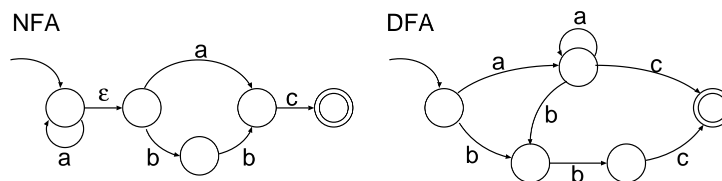


Figure 2.4. The non-deterministic and deterministic finite automata for the pattern `a*(a|bb)c`.

The difference between NFA and DFA is the necessity of backtracking; the former needs backtracking while the latter does not. Therefore, searching with DFA can be performed in $O(n)$ time where n is the length of the text. However, the transformation of NFA to DFA can be exponential to the length of the pattern. Hopcroft and Ullman[11] describes automata theory and Friedl[8] discusses practical applications of regular expressions. Kondou[32] presents an implementation guide for regular expression searching in C.

2.4. Incremental Searching

Incremental searching is a user interface for string searching widely used in text editors. According to Emacs manual[24], "an incremental search begins searching as soon as you type the first character of the search string. As you type in the search string, Emacs shows you where the string (as you have typed it so far) would be found".

Figure 2.5 shows the process of incremental search for 'scheme' with Emacs. As shown in the example, a user can quickly reach the position of 'scheme' with only four keystrokes [^S] [s] [c] [h] instead of typing [^S] [s] [c] [h] [e] [m] [e] with seven keystrokes.

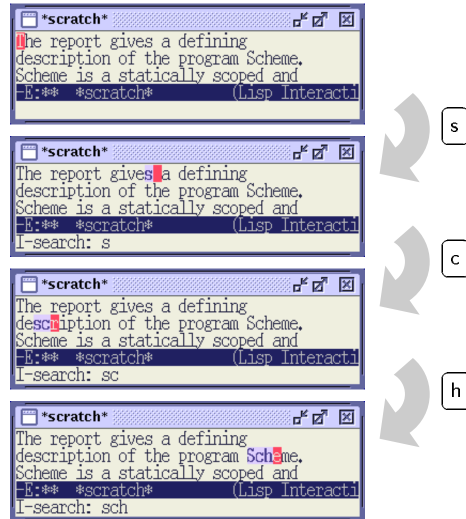


Figure 2.5. Process of incremental searching for 'scheme'.

Masui[35][36] integrated incremental searching and approximate string searching to realize incremental searching allowing errors. This method is called *dynamic approximate searching*.

2.5. Inverted File

An inverted file is a data structure for fast string searching. The structure is similar to the index attached to the end of books which allow readers to search for certain keywords quickly.

The inverted file consists of the *vocabulary* and the *occurrences*. The vocabulary is a set of all unique words in the text and the occurrences are positions where each word occurs in the text.

2.5.1 Construction of Inverted File

Suppose that we have a sample text "it is a bad bridge that is shorter than its stream" and wish to construct an inverted file for the text. The inverted file can be constructed with the following algorithm.

1. Read a word from the text.

2. If the word is not in the vocabulary, add it to the vocabulary with an empty occurrence list.

3. Add the position of the word at the end of its occurrence list.

4. Repeat the process until all words are scanned.

The algorithm can be easily implemented with modern programming languages which are capable of handling the two data structures: *list* and *hash table*.

Figure 2.6 illustrates the position of each word in the sample text “it is a bad bridge that is shorter than its stream”. The inverted file built on the sample text is depicted as Figure 2.7.

it	is	a	bad	bridge	that	is	shorter	than	its	stream
0	3	6	8	12	19	24	27	35	40	44

Figure 2.6. The sample text and the position of each word.

Vocabulary	Occurrences
a	6
bad	8
bridge	12
is	3, 24
it	0
its	40
shorter	27
stream	44
than	35
that	40

Figure 2.7. The inverted file for the sample text.

The occurrence field of the inverted file stores the positions of each word. It is also possible to store the additional information about document identifiers to construct an inverted file for a number of documents. For example, Web search engines employ the document identifiers to allow users find out the web pages containing certain keywords. Witten et al.[27] discussed and implemented a full-scale information retrieval system for millions of documents with the inverted file.

2.5.2 Searching with Inverted File

Single-word searching can be performed in two steps: searching the vocabulary and locating the occurrences. The method of vocabulary searching depends on how the vocabulary is represented. An efficient method is binary-searching which is realized by simply storing the vocabulary in lexicographical order. Phrase searching is performed as follows:

1. Search each word in the phrase separately.
2. Locate the occurrences for each word.
3. Merge the list of the occurrences according to the positions.

Usually, the vocabulary can be placed in ordinary sized main memory. Suppose the text has 500,000 unique words[1] and the average length of the words is 10, the vocabulary consumes about only 5 MB main memory.

2.5.3 Compression of Inverted File

Since occurrences are stored in ascending order, partial compression can be achieved by storing the positions in terms of relative distance. For example, occurrences of 1, 50, 293, and 349 can be represented as 1, 49, 243, and 56. Relative representation decreases the numbers so that occurrences can be encoded in shorter codes. A popular encoding for compressing positive integers of variable lengths is known as BER compression. Table 2.1 shows BER compressed 32 bit positive integers. The most significant bit (MSB) is set to 1 except the last byte. The 0 in MSB indicates termination of a integer. BER compression effectively represents small positive integers with short codes. For other advanced compression methods such as *Elias-*γ and *Golomb*, see Chapter 7 of [4].

Range of Integer	Bit Representation
`0x00000000-0x0000007f`	`0xxxxxxx`
`0x00000080-0x00003fff`	`1xxxxxxx 0xxxxxxx`
`0x00004000-0x001fffff`	`1xxxxxxx 1xxxxxxx 0xxxxxxx`
`0x00200000-0x0fffffff`	`1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx`
`0x10000000-0xffffffff`	`10000xxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx`

Table 2.1. BER compression for 32 bit positive integers.

[1] A large scale English dictionary Webster's Third New International Dictionary contains 460,000 entries.

2.6. Suffix Array

A suffix array is a data structure designed for efficient searching of a large text. The data structure consists of an array containing the indices to the text suffixes sorted in lexicographical order. Each suffix is a string starting at a certain position in the text and ending at the end of the text. Searching the text can be performed by binary search using the suffix array because the suffixes can be seen as the sorted suffixes in in the lexicographical order.

2.6.1 Construction of Suffix Array

Figure 2.8 through 2.10 explain a construction process of the suffix array for our sample text "it is a bad bridge that is shorter than its stream". First, we assign index points to the sample text. Index points specify positions where search can be performed. In our example, index points are assigned to word by word as shown in Figure 2.6. Thus, we can search the sample text with the suffix array at the beginning of each word. Second, we should sort the index points according to their corresponding suffixes. The correspondence between the index points and the suffixes is shown in Figure 2.8. Figure 2.9 shows the result of sorting. Suffixes are sorted in lexicographical order and index points are reordered according to their corresponding suffixes.

Index	Suffix
0	it is a bad bridge that is shorter than its stream
3	is a bad bridge that is shorter than its stream
6	a bad bridge that is shorter than its stream
8	bad bridge that is shorter than its stream
12	bridge that is shorter than its stream
19	that is shorter than its stream
24	is shorter than its stream
27	shorter than its stream
35	than its stream
40	its stream
44	stream

Figure 2.8. The suffixes for the sample text and their index points.

Finally, the resulting index points become the suffix array for the sample text as shown in Figure 2.10.

Index	Suffix
6	a bad bridge that is shorter than its stream
8	bad bridge that is shorter than its stream
12	bridge that is shorter than its stream
3	is a bad bridge that is shorter than its stream
24	is shorter than its stream
0	it is a bad bridge that is shorter than its stream
40	its stream
27	shorter than its stream
44	stream
35	than its stream
19	that is shorter than its stream

Figure 2.9. The suffixes sorted in lexicographical order. Each index point is reordered according to its corresponding suffix.

6	8	12	3	24	0	40	27	44	35	19

Figure 2.10. The suffix array for the sample text. Each entry corresponds to an index point shown in Figure 2.9.

2.6.2 Searching with Suffix Array

Searching of the sample text can be performed by binary search using the suffix array. Figure 2.11 shows the process of searching the sample text for ‘than.’ Numbered arrows show the order of the process.

The searching algorithm takes $O(P \log N)$ time where P is the length of the pattern and N is the length of the text. Manber and Myers[16] employs supplementary information about the longest common prefixes (lcps) to achieve $O(P + \log N)$. However, storing these information triples the size of the entire suffix array and both construction and searching algorithms become complicated. For applications using patterns shorter patterns, such as simple keyword searching, the factor of P can be negligible. Therefore, we do not employ the lcp information in this thesis.

2.6.3 Applications of Suffix Array

Applications of suffix arrays has been studied for years. In the field of natural language processing, the importance of effectively handling large corpora is crucial. Yamashita[38] proposed a method of morphological analysis using a suffix array. This method directly

	Index	Suffix
	6	a bad bridge that is shorter than its stream
	8	bad bridge that is shorter than its stream
	12	bridge that is shorter than its stream
	3	is a bad bridge that is shorter than its stream
	24	is shorter than its stream
1 ⟶	0	it is a bad bridge that is shorter than its stream
	40	its stream
	27	shorter than its stream
2 ⟶	44	stream
3 →	35	than its stream
	19	that is shorter than its stream

Figure 2.11. Searching the sample text for 'than' by binary search using the suffix array. Numbered arrows show the order of the process.

uses POS (part of speech) –tagged corpora as example-base instead of constructing a dictionary from the corpora. Itoh[13] proposed a similar example-based method for word segmentation. Bentley[5] presented an application of suffix array for generating sentences with Markov chain algorithm. In the field of computational biology, Gusfield[9] investigated applications for exploring genome database.

2.7. Comparison between Inverted File and Suffix Array

Table 2.2 summarizes the characteristics of inverted file and suffix array. Each issue is discussed in details in the following paragraphs.

	Inverted File	Suffix Array
Searching without the original text	yes	no
Construction time	fast	slow
Incremental update	easy	difficult
Compression	easy	difficult
Phrase searching	difficult	easy
Approximate searching	difficult	easy
Word-oriented indexing	easy	easy
Character-oriented indexing	difficult	easy

Table 2.2. Comparison between suffix array and inverted file.

- **Searching without the original text** An inverted file does not need the original text for searching while a suffix array needs. This means that the inverted file is more efficient in terms of space than the suffix array.

- **Construction time** An inverted file can be constructed in $O(n)$ time while a suffix array takes $O(n \log n)$ time. In other words, an inverted file can be constructed with sequential read of the original text while suffix array construction requires random accesses over the original text $O(n \log n)$ times.

- **Incremental update** When new texts are added, an inverted file can be updated efficiently by inserting newly added vocabulary and occurrences to the existing inverted file while a suffix array should be fully reconstructed.

- **Compression** An inverted file can be compressed efficiently while a suffix array is difficult to compress because the integers of index points is randomly ordered. Recently, Mäkinen[15] proposed *Compact Suffix Array* which solves this problem.

- **Phrase searching** An inverted file can be used for phrase searching which consists of two or more keywords such as "invisible computer" but the searching requires merging of lists of occurrences with high costs. In contrast, a suffix array can be used for phrase searching efficiently without additional processing.

- **Approximate searching** An inverted file can be used for approximate searching which allows errors in a query. However, errors crossing two or more words are hard to handle because the vocabulary is usually constructed as a set of single words. For example, approximate searching for "thebeatles" intended for "the beatles" is hardly performed even if a vocabulary list contains "the" and "beatles," respectively. In contrast, a suffix array can efficiently cope with approximate string searching by the method proposed by Yamashita[31].

- **Word-oriented indexing** An inverted file is suited for word-oriented indexing and a suffix array is also suited for word-oriented indexing. Since an inverted file is a word-oriented indexing method by nature, handling a language whose word segments are ambiguous such as Japanese requires word segmentation. Figure 2.12 shows the result of morphological analysis for Japanese sentence "真理は単純なものの中にひそむ" with ChaSen[19]. The leftmost column shows the segmented words and the others shows part of speech information for the words. The recall of searching using the inverted file highly depends on effective word segmentation. In other words, poor segmentation leads to the poor recall.

- **Character-oriented indexing** An inverted file is hardly suitable for character-oriented indexing while a suffix array is suited for character-oriented indexing as well as word-oriented indexing. A character-oriented inverted file can be constructed but it is not practical because the merging of occurrences takes a long time.

In summary, an invited file is suited for ordinary information retrieval task while a suffix array is suited for other text processing applications like corpus-based natural language processing which uses large corpora.

```
% echo '真理は単純なものの中にひそむ' | chasen
真理      シンリ       真理      名詞–一般
は        ハ           は        助詞–係助詞
単純      タンジュン   単純      名詞–形容動詞語幹
な        ナ           だ        助動詞 特殊・ダ 体言接続
もの      モノ         もの      名詞–非自立–一般
の        ノ           の        助詞–連体化
中        ナカ         中        名詞–非自立–副詞可能
に        ニ           に        助詞–格助詞–一般
ひそむ    ヒソム       ひそむ    動詞–自立 五段・マ行 基本形
EOS
```

Figure 2.12. Morphological analysis with ChaSen.

CHAPTER 3 *Our Search Systems*

A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away.
— Antoine de Saint-Exupéry

We have implemented three search systems: *Namazu*, *Sary*, and *Migemo*. Figure 3.1 shows the relation among our systems. Namazu is for searching vast quantities of documents, Sary is for a single large document, and Migemo is an interactive search interface.

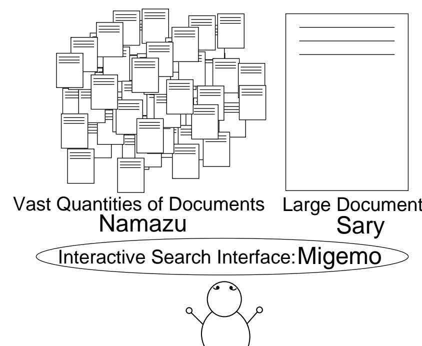


Figure 3.1. Relation among our systems.

3.1. Namazu

Namazu[21] is a full-text search engine intended for easy use. Not only does it work as a small or medium scale Web search engine, but also as a personal search system for email or other files. Namazu is freely available on the Internet as a free software.

Namazu was originally developed by the author of this thesis. However, the current implementation of Namazu has been developed by Namazu Project cooperatively including the author. Namazu is a successful free software in the term of popularity. In fact, Namazu version 2.x has been downloaded 28,861 times and used at a number of Web

sites, including a rising open source desktop environment GNOME[1] at the present time. Figure 3.2 shows the mailing list search engine using Namazu at the GNOME website.

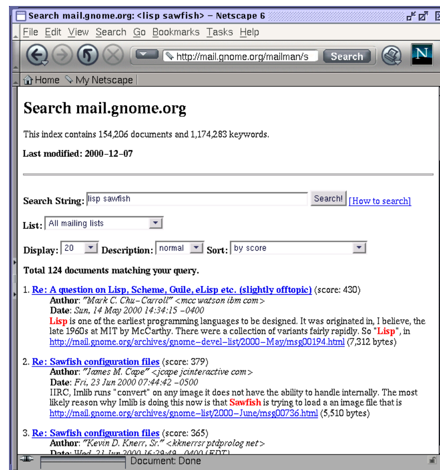


Figure 3.2. The mailing list search engine using Namazu at the GNOME website.

In technical point of view, Namazu is a simple search engine which employs an inverted file as the data structure. Characteristics of Namazu are summarized as follows:

- **Filters** Namazu can handle several types of document by filtering including: *plain text*, *HTML*, *email*, *PDF*, *Microsoft Word*, *Microsoft Excel*, *Microsoft Powerpoint*, *UNIX manual*, *TEX*. Some types of document requires external commands for filtering.

- **Fields** Fields like 'Subject:' line in email or '<title>' tag in HTML documents contain significant information. Namazu can index specific fields. A user can search for the email from a particular person by including 'From:' information in a query.

- **Interfaces** Namazu can be used with several interfaces including web browsers, editors, Tcl/Tk GUI, and command shells.

- **Handling of Japanese** Namazu is capable of handling Japanese documents. Namazu provides interfaces for ChaSen[19] or KAKASI[20] for word segmentation for Japanese texts.

[1] `http://www.gnome.org/`

Takabayashi[25] describes the system in detail. The author utilizes Namazu for the personal search engine. It is especially valuable for writing email for exchanging information with others.

3.2. Sary

Sary[26] is a suffix array library and tools. It provides fast full-text search facilities for a single large text on the order of 10 to 100 MB using a suffix array. It can also search specific fields in a text file by assigning index points to those fields. Sary is freely available on the Internet as a free software.

Another suffix array library SUFARY[30] has been available since 1997. The difference between Sary and SUFARY is the design philosophy. Sary is written in C but in object-oriented fashion. Sary aims for maintainability, extensibility, robustness, and usability rather than performance. In fact, the library of Sary does not use global or static variables at all except the version number for making all functions reentrant so that the library can be used in multi-threaded programming.

3.2.1 Block Sorting

Since suffix array construction requires $O((m+1)n)$ space where n is the size of the text and m is the size of an index point, processing a large text file in the main memory is sometimes not practical. $O((m+1)n)$ space consists of $1n$ for the original text and mn for the suffix array. With a 32 bit CPU, m is usually 4 bytes. For example, constructing a suffix array for 1 GB text requires 5 GB main memory because the original text consumes 1 GB and the suffix array consumes 4 GB.

Although modern operating systems support virtual memory which utilizes the hard disk as memory, suffix array construction in the main memory is highly desirable because page faults caused by virtual memory system reduces the construction performance. Accessing the hard disk is far slower than accessing the main memory.

To reduce the memory requirement, Sary provides a method of memory-saving external sorting. We call it *block sorting*. Block sorting can be defined as follows:

1. Split a large array of index points into small blocks.
2. Sort the partial blocks one by one.
3. Merge the sorted partial blocks to the single large suffix array.

Sorting of each block needs smaller memory than sorting the entire large array at once shot. We conduct an experiment that constructs a suffix array for a 35 MB text file with a machine equipping with only 64 MB main memory. The construction requires 175 MB memory. Table 3.1 shows that block sorting with the block size of 4 MB is much faster than normal sorting (i.e. sorting the entire array at one shot).

Sorting Method	Time (sec)
Normal Sorting	5359.28
Block Sorting	2118.12

Table 3.1. Suffix array construction with a memory-poor machine. The block size is set to 4 MB.

Since sorting of a block is independent to the other, each sorting can be performed in parallel by multi-threaded programming. Therefore, not only does block sorting save memory, but also speeds up the suffix array construction. The performance of parallel sorting is discussed in Section 3.2.3.

3.2.2 Merging

After block sorting is completed, the next stage requires merging of sorted partial blocks. A simple algorithm is the binary merging as shown in Figure 3.3. Let n be the number of total index points and m is the number of sorted partial blocks, the number of phases for partial merging can be estimated at $O(\log m)$. Since the cost of each phase is $O(n)$, the whole merging process can be completed in $O(n \log m)$ time.

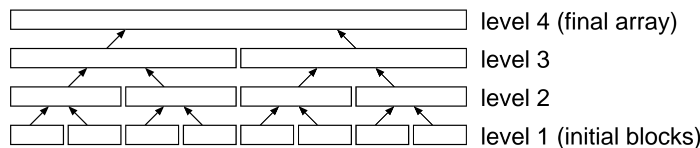


Figure 3.3. Merging the sorted partial blocks in a binary fashion.

Another algorithm employs a priority queue for extracting the minimum element from all the sorted partial blocks one by one and writes the resulting suffix array sequentially. Sary uses a heap structure for representing a priority queue as shown in Figure 3.4.

With the heap structure, extracting the minimum element can be performed in $O(1)$ time. However, the heap should be rearranged at each extraction that takes $O(\log m)$

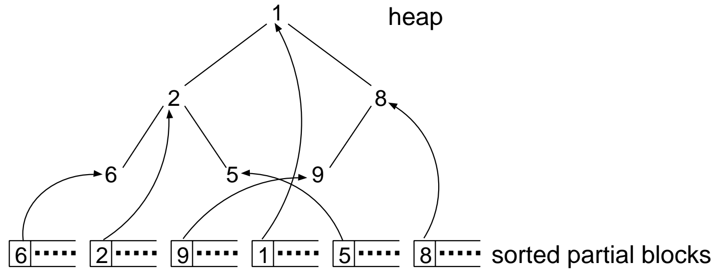


Figure 3.4. Merging the sorted partial blocks with the heap structure.

where m is the number of sorted partial blocks. Therefore, merging can be completed in $O(n \log m)$ time where n is the number of total index points.

Although the former algorithm (i.e. binary merging) could be parallelized in each partial merging phases while the latter could not, we employ the latter algorithm because the latter is faster than the former on the simple experiment. Theoretically, parallelized binary merging can be very efficient if plenty of processors are used. However, as discussed above, increasing the number of threads makes the performance poor even though the machine has plenty of CPUs.

3.2.3 Performance

Construction Time

We conduct experiments for constructing a suffix array for a 80 MB text file with block sorting at several block sizes. The experiments are performed with three platforms as shown in Table 3.2. Although the Alpha and MIPS machines have four and 32 CPUs respectively, increasing the number of threads makes the performance poor. We assume this is because of limit of the memory bus. Thus, the experiment is completed with two and four threads respectively to bring out their best performance.

The Figure 3.5 show the results. The graphs on the left show the results without multi-threading and the graphs on the right show the results with multi-threading. Horizontal lines within these graphs indicate performance of normal sorting (i.e. not block sorting) for comparison.

The result shows that multi-threaded block sorting is faster than normal sorting for large text files. Note that Sary employs *multikey quicksort*[6] as the sorting algorithm. Itoh[33] proposed an even faster suffix array construction algorithm.

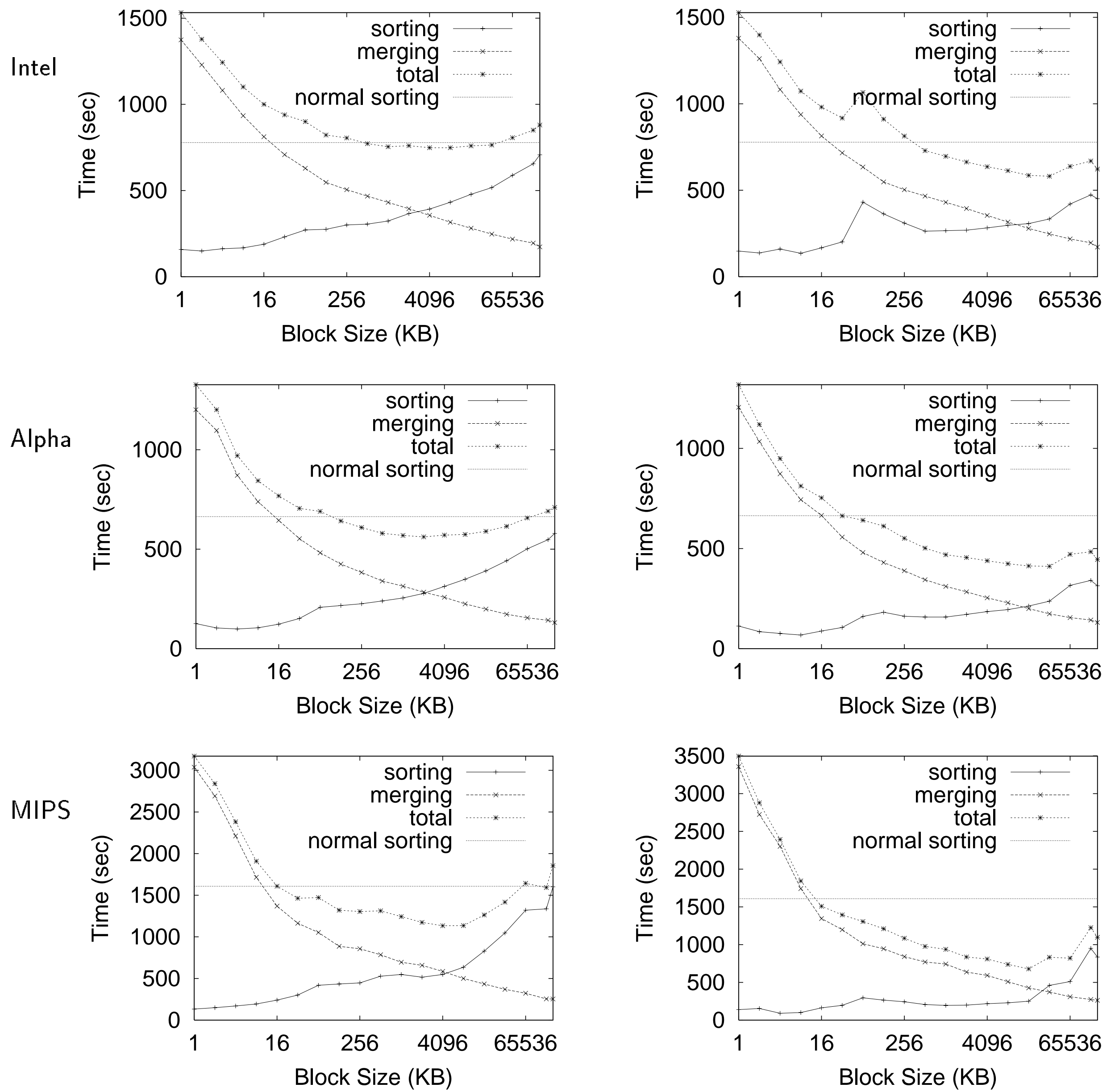


Figure 3.5. Suffix array construction with the Intel, Alpha, and MIPS machines. The right graphs shows the results with multi-threading.

CPU	Memory	OS
Intel Pentium III 600 MHz × 2	2 GB	Red Hat Linux 6.2J
Alpha 21164A 466 MHz × 4	2 GB	Digital UNIX V4.0F
MIPS R10000 195 MHz × 32	5 GB	IRIX 6.4

Table 3.2. Platforms used for the experiments on suffix array construction.

Searching Time

Searching performance is hard to measure accurately because modern operating systems have disk caches. We conduct an experiment searching a 200 MB text with a machine having 2 GB main memory. 2 GB memory allows loading the entire file in the disk cashes so that disk accesses never occur. The experiment was performed with the Alpha machine. The result shows that sary command takes only 0.04 second for searching the 200 MB text while grep command takes 11 second on the average.

3.3. Migemo

Migemo is a user interface which allows users to find information quickly with fewer keystrokes. Although text editors have evolved, facilities for searching have merely improved. In fact, most editors cannot search Japanese characters efficiently. For example, to search for '活発' with Emacs editor, users have to type at least 11 keystrokes: [^S] [KANA] [k] [a] [p] [p] [a] [t] [u] [CONVERT] [KANA] because of the necessity of Kana-Kanji conversion. [KANA] key starts and ends Kana input and [CONVERT] key converts the Kana string to Kanji string. As in the example, [CONVERT] converts 'かっぱつ' (Kana string) to '活発' (Kanji string). The necessity of Kana-Kanji conversion inhibits incremental searching for Japanese words. People have to type the entire spelling of a word and convert it to Kanji for inputting the Kanji word. It highly annoys. Incremental searching is indispensable for quick searching.

Some Japanese input methods such as T-Code[1] and TUT-Code[37] provide the facility for direct Japanese input but they are not popular because remembering key combinations of thousands of Kanji characters is hard for casual users.

Migemo realized Japanese incremental searching by expanding a pattern dynamically into regular expressions representing a set of Japanese words as a user types in the search string. We call this method *dynamic pattern expansion*. Figure3.6 shows the process that a user reaches the position of '活発' with only four keystrokes: [^S] [k] [a] [p] without

Kana-Kanji conversion.

Figure 3.6. Process of incremental searching for '活発' with Migemo.

3.3.1 Dynamic Pattern Expansion

Dynamic pattern expansion is a method for realizing advanced incremental searching. In the application of Migemo, the dynamic pattern expansion employs an ordinary Kana-Kanji dictionary. Figure 3.7 illustrates the process of sequential pattern expansions for 'k', 'ka', and 'kap'. The expanded pattern for 'k' takes 5,085 bytes and the pattern for 'ka' takes 2,140 bytes. Both are large regular expressions matching many positions in the text. However, as incremental search proceeds, the expanded pattern shrinks gradually and positions where the pattern matches narrows simultaneously. Thus, the expanded pattern for 'kap' takes only 158 bytes.

In Japanese, one Kana representation may have different Kanji representations. For example, Kana representation 'きかい' (kikai) has various Kanji representations such as '機械', '機会', '奇怪' and '器械'. Kana-Kanji conversion requires the user to select the Kanji representation which he or she wants from the candidates. The cognitive load of the Kanji selection is very high if there are a number of candidates. One advantage of Migemo is that it does not require Kanji selection. Obviously, Migemo can find an inappropriate text position because of the diversity of Kanji representations. However, such

k

[k 、。,. ・:;?!゛゜´｀¨＾ ＿ヽヾゝゞ〃仝々〆〇ー―‐／＼～‖…
‥‘“”（）〔〕［］｛｝〈〉《》「」『』【】＋－±×÷＝≠＜＞≦≧∞
∴♂♀°′″℃￥＄￠￡％＃＆＊＠§☆★○●◎... (5,085 bytes)

ka

[～‘“”（）〔〕［］｛｝〈〉《》「」『』【】×＊→⊃∧∠かくカケヵ
ヶΚκχ┌┐┘└哀愛囲易畏一蔭陰隠烏渦影鴎下化仮何伽価佳加可
嘉夏嫁家寡科暇果架歌河火珂禍禾稼箇花苛茄... (2,140 bytes)

kap

[Κκ]|kap|かっ|かぱ|かぴ|かぷ|かぺ|かぽ|カッ|カパ|カピ|カプ|カペ
|カポ|河童|割賦|割腹|割烹|喝破|恰幅|活溌|活発|活版|合羽|勝平|川童|
闊歩|濶歩|ｋａｐ|Ｋａｐｐａ

Figure 3.7. The sequential dynamic pattern expansions for for ‘k’, ‘ka’, and ‘kap’.

kind of situations rarely occur if the user types five or more keystrokes. Even though the inappropriate matching occur, the user can simply ignore it with ^S key for proceeding to the next matching position.

Some people may worry about the cost of the dynamic pattern expansion. In fact, the cost is high and Migemo employs pre-compiled patterns for frequent short strings for reducing the cost. However, since the performance of computers has been improved dramatically and improvement is likely to continue, the overhead will be negligible in near future. Masui[34] proposed the approach called “Fugou-teki Programming” (millionaire programming) which uses resources massively for creating a better software.

The author has been using Migemo a half year in comfort. Comments from other users is also positive. However, quantitative evaluation is not conducted yet.

3.3.2 Applications

The application of Migemo is not only limited to incremental searching for Japanese, but for any languages using front-end processing for inputting a word such as Chinese. The technique of the dynamic pattern expansion can be applied to various situations in which people have difficulty in incremental searching. All sorts of dictionaries can be used for the dynamic pattern expansion. For example, English-Japanese dictionary can be used for incremental searching. Figure 3.8 shows the process of searching a Japanese word ‘情報’ with an English word ‘information’. It is useful for English speakers who are learning Japanese. Figure 3.9 shows the application which employs a biographical dictionary to

search for a person's name.

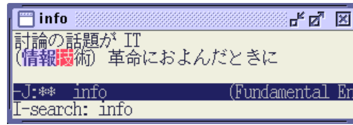


Figure 3.8. Searching a Japanese word '情報' with an English word 'information'.

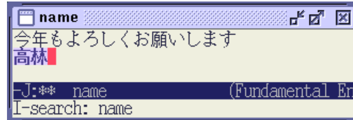


Figure 3.9. Conceptual searching for a person's name.

CHAPTER 4 *Input Acceleration*

Don't use hands to do things that can be efficiently done by the computer.
— Tom Duff

Typing with a keyboard is not always the best way to input sentences. For example, people can do copy-and-paste for reusing a previously-input sentence instead of retyping. Inputting Japanese sentences has difficulties which English does not have, for example, the necessity for Kana-Kanji Conversion. In this chapter, we discuss several methods for input easily and accelerated input.

4.1. Dynamic Abbreviation

Dynamic abbreviation is a technique for input acceleration. It saves the keystrokes for inputting previously-input words. For example, if a user wants to input 'floccinaucinihilipilification[1]' for the second time, dynamic abbreviation allows the user to input 'floccinaucinihilipilification' with short keystrokes. Figure 4.1 illustrates the process of dynamic abbreviation with Emacs editor. The user types only two keystrokes: f META-/. META-/ key triggers the dynamic abbreviation to input 'floccinaucinihilipilification'. It saves as many as 27 keystrokes.

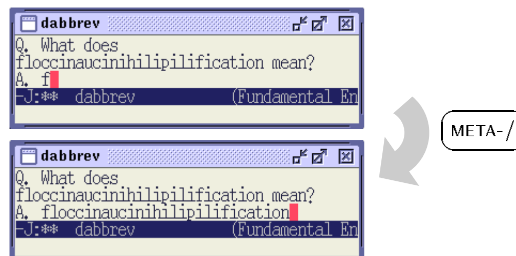


Figure 4.1. Dynamic abbreviation for 'floccinaucinihilipilification'.

The mechanism of the dynamic abbreviation is fairly simple. As in the example, the

[1] It means "the estimation of something as worthless."

dynamic abbreviation first search for the word which begins with 'f' backward and then insert the word into the point where the cursor stays.

A problem arises for performing dynamic abbreviation for Japanese words because of the necessity of Kana-Kanji conversion. For example, performing the dynamic abbreviation for '奈良先端科学技術大学院大学' requires the user to type eight keystrokes: KANA n a r a CONVERT KANA META-/. The Kana-Kanji conversion problem arises again.

To solve this problem, we propose a method for performing dynamic abbreviation for Japanese words without Kana-Kanji conversion. Our method employs dynamic pattern expansion discussed in Section 4.1. Figure 4.2 illustrates the process of dynamic abbreviation for '奈良先端科学技術大学院大学'. As in the example, the user can input '奈良先端科学技術大学院大学' with only three keystrokes: n a META-/. This process employs Migemo to search for the word which begins with 'na' backward. The rest of processing is identical to ordinary dynamic abbreviation.

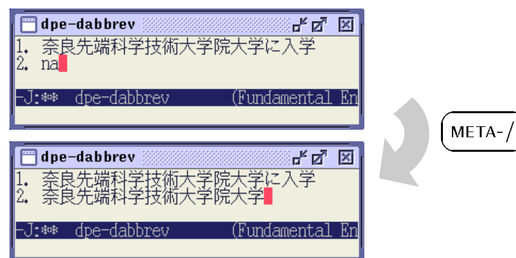


Figure 4.2. Dynamic abbreviation for '奈良先端科学技術大学院大学'.

4.2. Quick Copy-and-Paste

In the field of software engineering, importance of reusing programs is highly emphasized. We think that reusing of information in technical writing is equally important. For example, people frequently consult a number of documents written by themselves to compose new documents and copy-and-paste the previously-written sentence in their current works instead of composing a new sentence from scratch. Masui[17] argued that "in some cases, it is faster to search an existing text close to the requirement and make modifications according to needs rather than writing the text from scratch."

The current copy-and-paste process takes the following three steps:

1. Search documents.

2. Selecting an appropriate sentences/phrases/expressions.

3. Copy-and-Paste.

This process requires many operations that make people exhausted. We propose a method to reduce obstacles of reusing information. Our method integrate the three steps into a single step by employing a suffix array and dynamic pattern expansion. The suffix array is used for fast string searching and the dynamic pattern expansion allows the user to input search string without Kana-Kanji conversion.

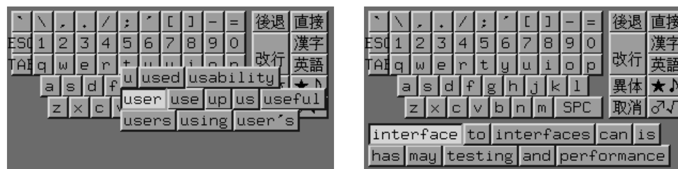
Figure 4.3 illustrates a typical process of reusing information with our system. A user is supposed to wishes to tell a friend information about a recently-published programming book and wants to reuse the sentence written yesterday for another purpose. The user can find the latest sentence mentioning about programming and paste it quickly. Then, the user can edit the old sentence to compose a new message. The process is smoothly integrated so that the user feels no frustration.

4.3. Input Prediction

Masui[18] proposed an efficient text input method called *POBox* which allows a user to input words with fewer keystrokes. It is especially useful for handheld computers which are not equipped with efficient text input devices like a keyboard. Figure 4.4 depicts the snapshots of pen-based POBox. When the user taps 'U' key, POBox presents frequently used words beginning with 'U' as candidates. And after the user selects 'user', POBox predicts the next word and presents candidates. The right picture in Figure 4.4 shows that 'interface' is the first candidate because 'interface' is likely to follow the word 'user'.

Although input prediction can be a very effective way to input words, prediction of the next word is a difficult task. One approach to solve the task is to construct a dictionary for prediction with n-gram model. However, n-gram model limits its prediction capability to n words. We propose a method for input prediction by searching previously-written documents as the corpus instead of constructing the dictionary. For example, if an ardent reader of "Structure and Interpretation of Computer Programs"[2] types 'Structure and,' the following phrase would be the most likely 'Interpretation of Computer Programs.' Such kind of prediction can be easily performed by dynamically searching the previously-written documents. Figure 4.5 illustrates the process of inputting 'Structure and Interpretation of Computer Programs' with our system. The user can input the phrase quickly by searching and reusing the previously input phrase.

Figure 4.3. Process of quick copy-and-paste. The user reuses information about a recently-published book.



After tapping 'U' key After selecting 'user'.

Figure 4.4. Inputting 'user interface' with Pen-based POBox.

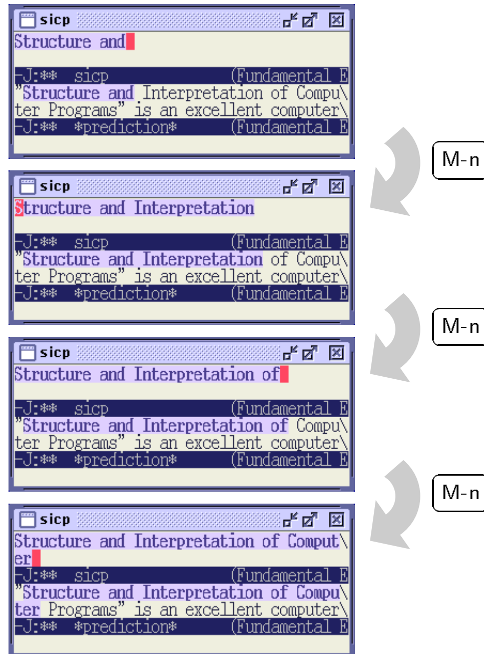


Figure 4.5. Process of input prediction with the previously written sentence.

The implementation of the system is almost identical to the system discussed in Section 4.2. Since our system is in the early stage of development, the current implementation uses only raw texts. Linguistic knowledge like morphological information can also be employed for more accurate prediction.

CHAPTER 5 *Proofreading*

Example is not the main thing in influencing others. It is the only thing.
— Albert Schweitzer

Writing an error-free document is necessary for expressing ideas without being misunderstood by readers, but it is difficult especially for a non-native writer. For example, many Japanese writers have difficulty in writing a document in English. Writers can use dictionaries to find sample sentences. However, they hardly find the exact sentence they want, since sample sentences in dictionaries are not abundant. In this chapter, we present a method to solve the problem by employing large corpora as example-base. We call this approach *Proofreading by Example.*

5.1. Finding Sample Sentences

Writers need enough sample sentences, especially to write technical papers efficiently. Currently, millions of documents written by native writers are available on the Internet. As of January 18, 2000, Inktomi Corp. and the NEC Research Institute[12] have reported that the Web has grown to more than 1 billion different pages.

Furthermore, papers in the specific fields are also available and it is possible to utilize the papers as sample sentences. One such source which can be available freely is e-Print archive[1]. The website offers papers in a variety of fields.

Consequently, utilizing a number of papers as sample sentences is realistic, but the problem is how to explore such a large volume of papers. People would be frustrated when searching for a sentence takes too long even if they have 100 MB of sample sentences. To reduce the search time, we should employ certain data structures. Since finding sample sentences requires the efficient phrase searching, we employ a suffix array instead of an inverted file.

To utilize a number of papers as sample sentences with the suffix array, one should concatenate the papers to one large text file and construct the suffix array for the con-

[1] <http://arXiv.org/>

catenated file. Concatenating the papers loses identity of documents. However, it is not a problem because information about which document contains a sentence is not important in the task of finding sample sentences. Having constructed the suffix array, searching of the papers can be performed very fast. Figure 5.1 shows the snapshot of our system within the Emacs editor. Our system dynamically finds sample sentences as the user types search strings just like incremental searching. The response time is fast enough thanks to the suffix array. Phrases can also be searched similarly as shown in Figure 5.2.

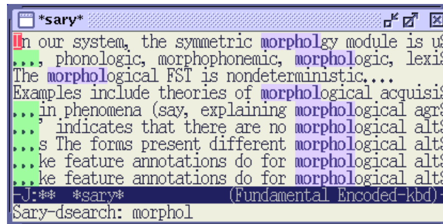


Figure 5.1. Finding sample sentences for 'morphological'. The user only types 'morphol' at the moment.

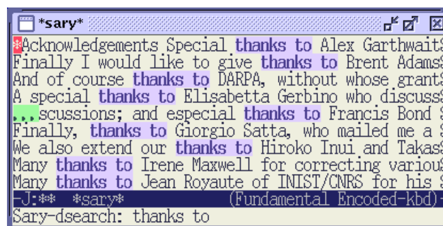


Figure 5.2. Finding sample sentences for 'thanks to'.

Although the current implementation of our system is very useful, there are various problems which should be solved in future.

First, our system does not support proximity searching which allows gaps in a phrase. For example, our system does not work well to find sample expressions such as 'look *something* up'. Proximity searching can be performed with a suffix array by searching 'look' and 'up' separately and then merging their occurrence lists with a certain proximity.

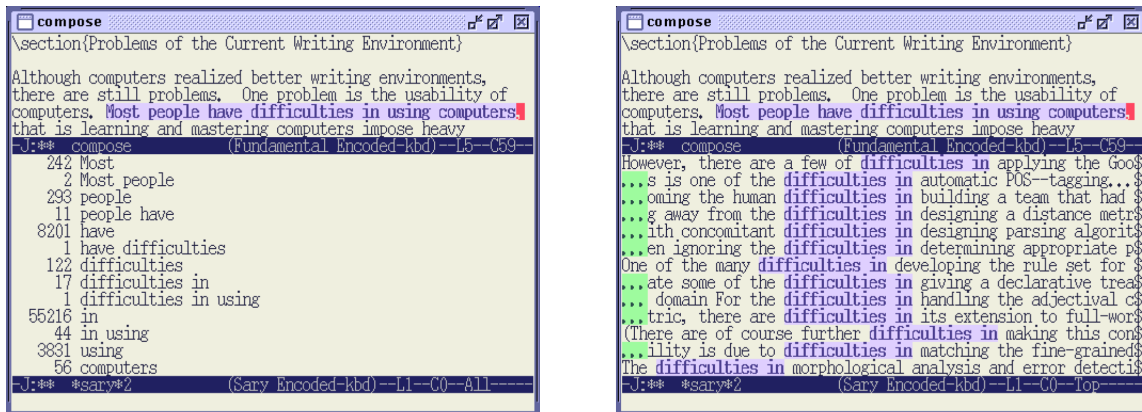
Second, conjugated verbs such as 'be,' 'is,' 'am,' 'are,' 'was,' and 'were' should be identified. It is desirable that the user can find sample expressions such as 'is scored' and 'are scored' simultaneously by simply typing '*be* scored'. Similarly, different tense and countability should be dealt. These problem can be solved by supporting regular expression searching. For example, it is possible to expand 'be' to the regular expression

pattern '(is|am|are|was|were)' using the method of dynamic pattern expansion discussed in Section 3.3.1.

Third, there are problems which cannot be solved only with the raw text. For example, a user may want to find sample sentences for some ambiguous words like for 'take' as a noun. Such a problem requires morphological information. One solution to the problem is that the system constructs the suffix array for morphological information as well as the suffix array for the surface text.

5.2. Checking Expressions

Non-native writers often worry about which expression to select in writing. One way to check whether an expression is natural or not is to consult sample sentences as discussed in Section 5.1. We present a more efficient way to check the expressions. Our method employs the documents written by native writers as a corpus and check the expressions by comparing with the corpus. Figure 5.3 illustrates checking of an expression 'Most people have difficulties in using computers' with our system. The left window shows the frequencies of the phrases in the expression. With such information, the user can check the expression statistically. If the frequency of a phrase in the expression is high, the phrase is likely to be natural. The right window shows the finding of sample sentences for 'difficulties in' with the system discussed in Section 5.1. Checking an expression and finding sample sentences are seamlessly integrated. The method can be extended to a spell checker if the corpus with no typos are provided.



Consult the corpus for frequencies of phrases. Finding sample sentences for 'difficulties in' seamlessly.

Figure 5.3. Checking the expression 'Most people have difficulties in using computers' with our system.

There are problems which cannot be solved only with surface information of the text. For example, a user may want to check an expression of ‘this are pens’ which is not a valid sentence while the system tells the user the frequencies of ‘this are’ and ‘pens’ are 12 and 4, respectively. Thus, the user may believe the expression is natural. However, the expressions ‘this are’ in the corpus are used like ‘Good examples of this are’ and that does not support ‘this are pens’ is a natural expression. Figure 5.4 illustrates the situation. To solve this problem, advanced linguistic knowledge like grammatical construction is required.

The system shows the frequency of ‘this are’ is 12.

‘this are’ in the corpus are used in appropriate ways.

Figure 5.4. Checking the expression ‘this are pens’

CHAPTER 6 *Conclusions and Future Work*

Who reflects too much will accomplish little.
— Friedrich von Schiller

This thesis has presented various applications of search techniques for writing assistance in terms of our methodology "Writing is Searching". Chapter 1 discussed the problems of the current writing environment. Chapter 2 described several search techniques and explained characteristics of the techniques. In Chapter 3, we presented the implementation issues of our search systems: *Namazu*, *Sary*, and *Migemo*. The systems laid the foundation of the works in the following chapters. In Chapter 4, we proposed three methods for input acceleration by employing dynamic string searching of large corpora: dynamic abbreviation for Japanese words, quick copy-and-paste for reusing information, and input prediction for phrases. Chapter 5 proposed two methods for proofreading: finding sample sentences for technical writing and checking expressions. Both methods employ the documents written by native writers as example-base. We called the approach Proofreading by Example.

While the author believes that these techniques help people to write technical documents substantially, there are many scopes of improvements. First, the searching of the raw text limits the applications of writing assistance. There are problems which cannot be solved with the raw text. Linguistic knowledge like morphological information should be incorporated. Second, fast string searching based on a suffix array lacks flexibility for handling today's information flow because the suffix array can not be updated efficiently. One solution to the problem is using the suffix array for searching static information only and using another search techniques for dynamic information. Finally, more effective assistance should be achieved. One possibility is an application of Just-In-Time Information Retrieval[22][23] which proactively retrieves and presents information based on a person's local context.

For worldwide communication, further research on writing assistance should be carried out. This work is only the first step to our future research project.

Acknowledgements

My deepest gratitude goes to Professor Yuji Matsumoto for his supervision. In addition to giving me good insights into this work, his constant encouragement helped me very much. I thank the second supervisor Professor Katsumasa Watanabe for carefully reading this thesis.

I like to thank Associate Professor Yasuharu Den who generously guided me around Stanford University on my vacation of the first year. I acknowledge Dr. Takashi Miyata's valuable advice and help in solving various problems.

I am also grateful to Dr. Toshiyuki Masui for sharing his experiences with his precious suggestions. His works are the inspiration of this work. He also generously invited me Sony CSL Open House and gave me interesting talks.

My first year at NAIST was entirely satisfactory because of the firm friendship with Tatsuo Yamashita. I learned a great deal about algorithms and data structures as well as everyday life wisdom from him. His research laid the groundwork for this work.

I wrote a number of programs during this work. My programming skill is owed to Masatake Yamato and Mitsuru Oka. They sincerely taught me real programming techniques and methodology. They are truly excellent programmers.

Many thanks go to Yuuta Tsuboi who suggested me to insert quotations every chapter and Kazuma Takaoka who kindly helped me out from the TEX nightmare. They were always good company when I wanted to escape from writing.

I would also like to thank Kou Kawabe and Daichi Mochihashi for their deep understanding of computer science and philosophical issues. Discussion with them enlightened me highly.

Thanks to Maruf Hasan and Kaoru Yamamoto for carefully reading this thesis and giving me many valuable comments for English writing as well as research.

A special thanks is reserved for Kazuhiko Shiozaki. This thesis is typesetted using his refined *efont-serif* font.

Last but not least, thanks to all staffs and members of Matsumoto laboratory. They provided supportive environment all the time and tolerated me patiently. I will never forget the well-spent two years at NAIST.

References

[1] T-Code laboratory. http://openlab.ring.gr.jp/tcode/.

[2] Hal Abelson, Jerry Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1999.

[3] Jun'ichi Aoe, editor. *Computer Algorithms String Pattern Matching Strategies*. IEEE Computer Society Press, 1994.

[4] Ricardo Baeza-Yates and Berthier Rieiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[5] Jon Bentley. *Programming Pearls*. Addison-Wesley, second edition, 2000.

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

[7] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, Vol. 20, pp. 762–772, 1977.

[8] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly, 1997.

[9] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. IEEE Computer Society Press, 1997.

[10] Patrik A. V. Hall and Geoff R. Dowling. Approximate string matching. *ACM Computing Surveys*, Vol. 12, No. 4, pp. 381–402, 1980.

[11] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[12] Inktomi Inc. Web surpasses one billion documents, January 2000. http://www.inktomi.com/new/press/2000/billion.html.

[13] Hideo Itoh. A method for segmenting japanese text into words by using suffix array. In *IPSG SIG-NL 99-NL-131*, pp. 47–54, 1999.

[14] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, Vol. 6, No. 2, pp. 323–350, 1977.

[15] Veli Mäkinen. Compact suffix array. In *The 11th Annual Symposium on Combinatorial Pattern Matching*, pp. 305–319, June 2000.

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

[17] Toshiyuki Masui. Integrating pen operations for composition by example. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 211–212, 1998.

[18] Toshiyuki Masui. POBox: An efficient text input method for handheld and ubiquitous computers. In *Proceedings of the International Symposium on Handheld and Ubiquitous Computing*, pp. 289–300, September 1999.

[19] Yuji Matsumoto, Akira Kitauchi, Tatsuo Yamashita, Yoshitaka Hirano, Hiroshi Matsuda, and Masayuki Asahara. Japanese morphological analysis system chasen version 2.0 manual 2nd edition, 2000. http://chasen.aist-nara.ac.jp/.

[20] KAKASI Project. Kakasi - kanji kana simple inverter. http://kakasi.namazu.org/.

[21] Namazu Project. Namazu: a full-text search engine, 2000. http://namazu.org/.

[22] Bradley J. Rhodes. Remembrance agent. In *The Proceedings of The First International Conference on The Practical Application Of Intelligent Agents and Multi Agent Technology*, pp. 487–495, 1996.

[23] Bradley J. Rhodes. *Just-In-Time Information Retrieval*. PhD thesis, Massachusetts Institute of Technology, 2000.

[24] Richard Stallman. *GNU Emacs Manual*. 2000.

[25] Satoru Takabayashi. Namazu: Yet another full-text search engine for vast quantities of documents. *IPSJ Magazine*, Vol. 41, No. 11, pp. 1227–1232, 2000.

[26] Satoru Takabayashi. Sary: a suffix array library and tools, 2000. http://sary.namazu.org/.

[27] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, second edition, 2000.

[28] Sun Wu and Udi Manber. Agrep - a fast approximate pattern-matching tool. In *Proceedings of USENIX Technical Conference*, pp. 153–162, January 1992.

[29] Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, Vol. 35, No. 10, pp. 83–91, 1992.

[30] Tatsuo Yamashita. SUFARY home page. http://cl.aist-nara.ac.jp/~tatuo-y/ss/.

[31] Tatsuo Yamashita and Yuji Matsumoto. Full text approximate string search using suffix arrays. In *IPSJ SIG-NL 97-NL-121*, pp. 83–90, 1997.

[32] 近藤嘉雪. 定本 C プログラマのためのアルゴリズムとデータ構造. ソフトバンク, 1998.

[33] 伊東秀夫. 文字列索引法とその自然言語処理への応用. PhD thesis, 東京工業大学, 2000.

[34] 増井俊之. 富豪的プログラミング. *bit*, Vol. 29, No. 1, pp. 36–37, 1997.

[35] 増井俊之. 動的曖昧検索. *UNiX MAGAZINE*, Vol. 13, No. 1, pp. 65–69, 1998.

[36] 増井俊之. 動的曖昧検索プログラムの作成. *UNiX MAGAZINE*, Vol. 13, No. 2, pp. 76–87, 1998.

[37] 大岩千穂子. TUT-Code homepage. http://www.crew.sfc.keio.ac.jp/~chk/.

[38] 山下達雄. 品詞タグ付きコーパスを直接利用した形態素解析. 言語処理学会第 4 年次大会予稿集, pp. 524–527, 1998.