NAIST-IS-DT0161025

Doctor's Thesis

Synthetic Assistance for Creation and Communication of Information

Satoru Takabayashi

February 6, 2004

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

Doctor’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
DOCTOR of ENGINEERING

Satoru Takabayashi

Thesis committee: Yuji Matsumoto, Professor
Shunsuke Uemura, Professor
Minoru Itoh, Professor

Synthetic Assistance for Creation and Communication of Information*

Satoru Takabayashi

Abstract

As the Internet becomes popular, circulation of information increased rapidly and many researchers have actively studied how to create and share information effectively. Creation and sharing of information can be seen as a continuous process of 1) how to find necessary information, 2) how to arrange the information, and 3) how to share the information with others. We consider that search and communication are especially important activities in the process. The goal of this thesis is more effective creation and sharing of information through the effective use of search and communication techniques.

First, we propose an incremental search method for Japanese text called *Migemo* that improves usability of search operations, and show how Migemo effectively improves the search operations through the evaluation conducted with subjects who use Migemo regularly. Second, we propose several methods for writing assistance through various search techniques, and show how the methods help writing substantially. Finally, we propose a group communication system called *QuickML* that allows users share information with others easily, and show how QuickML effectively help users enjoy casual group communication through the public service involving thousands of users.

In this way, we believe that circulation of information on the Internet increased much more substantially by supporting creation and sharing of information as a continuous process.

*Doctor's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT0161025, February 6, 2004.

Keywords:

Search, Communication, Writing Assistance, Instant Group Communication, Incremental Search Method for Japanese Text, Suffix Array, Migemo, QuickML, Namazu.

Contents

1 Introduction **1**

1.1 Search and Communication . . . 2

1.2 Approach . . . 2

1.3 Organization of the Thesis . . . 3

2 Search Techniques **5**

2.1 Introduction . . . 5

2.2 Sequential String Searching . . . 5

2.2.1 Approximate String Searching . . . 6

2.2.2 Regular Expression Searching . . . 8

2.3 String Searching using Indices . . . 9

2.3.1 Inverted File . . . 9

2.3.2 Suffix Array . . . 12

2.3.3 Comparison between Inverted File and Suffix Array . . . 15

2.4 Implementations of Our Search Systems . . . 17

2.4.1 Namazu: A Full-Text Search Engine . . . 18

2.4.2 Sary: A Suffix Array Library and Tools . . . 19

2.5 Summary . . . 24

3 Incremental Search Method for Japanese Text **25**

3.1 Introduction . . . 25

3.2 Migemo . . . 28

3.2.1 Method . . . 28

3.2.2 Generation of Regular Expressions . . . 29

3.2.3 Implementations . . . 32

3.3 Evaluation . . . 33

3.4 Discussion . . . 34

3.4.1 Other Approaches . . . 34

3.4.2 Handling of Homonyms . . . 35

3.4.3 Handling of Multi-Segment Phrases 36
3.4.4 Use of a Dictionary 37
3.5 Applications 37
3.5.1 Sharing a Dictionary with an Input System 38
3.5.2 Language-Independent Incremental Search 38
3.6 Summary 39

4 Writing Assistance through Search Techniques 41
4.1 Introduction 41
4.2 Problems of the Current Writing Environment 42
4.2.1 Our Methodology: “Writing is Searching” 44
4.3 Input Acceleration 44
4.3.1 Dynamic Abbreviation 44
4.3.2 Quick Copy-and-Paste 46
4.3.3 Input Prediction 48
4.4 Proofreading 50
4.4.1 Finding Sample Sentences 50
4.4.2 Checking Expressions 52
4.5 Summary 54

5 Instant Group Communication 55
5.1 Introduction 55
5.2 Problems of Conventional Mailing Lists 55
5.3 QuickML 57
5.3.1 Usage of QuickML 57
5.3.2 Effective Uses 61
5.3.3 Disadvantages 63
5.4 Implementation 63
5.4.1 Automatic Administration of Mailing Lists 64
5.4.2 Using Agents with QuickML 66
5.5 Discussion 67
5.5.1 Experiences 67
5.5.2 Statistical Observations 69
5.6 Related Work 71

5.7 Summary 73

6 Conclusion **75**

References **77**

Acknowledgements **83**

List of Publication **84**

List of Figures

2.1 Three types of errors in a pattern. 6
2.2 Dynamic programming for searching a text "failure" for a pattern "figure". The lower right element indicates that the search is successful with two errors. 7
2.3 The NFA for a pattern "figure" with two errors. The shaded states and the bold lines illustrate the successful transition of reading a text "failure". 8
2.4 The non-deterministic and deterministic finite automata for a pattern `a*(a|bb)c`. 9
2.5 The sample text and the positions of each word. 10
2.6 The inverted file for the sample text. 10
2.7 The suffixes for the sample text and their index points. 13
2.8 The suffixes sorted in a lexicographical order. Each index point is reordered according to its corresponding suffix. 13
2.9 The suffix array for the sample text. Each entry corresponds to an index point shown in Figure 2.8. 14
2.10 Searching the sample text for "than" by binary search using the suffix array. Numbered arrows show the order of the process. 14
2.11 Morphological analysis with ChaSen. 17
2.12 The mailing list search engine using Namazu at the GNOME website. 18
2.13 Merging sorted partial blocks in a binary fashion. 21
2.14 Merging sorted partial blocks with a heap structure. 21
2.15 Suffix array construction with Alpha and MIPS machines. The right graphs show the results with multi-threading. 23

3.1 A common dialog for keyword search. 26
3.2 Incremental search for "scheme". 26
3.3 The process of selecting "奇怪 (strange)" as a search keyword. 27
3.4 Incremental search for 「奇怪 (kikai)」 using Migemo. 28

3.5 Generated regular expressions for “k”, “ki”, and “kik”. 29
3.6 The dictionary for generating regular expressions (excerpt). 30
3.7 Optimization of the regular expression for “mour”. 30
3.8 Migemo for Emacs. 32
3.9 Migemo implementations: w3m, VIM, and xyzzy. 33
3.10 Homonyms occurred in the text of a Japanese paper on Migemo. . . . 36
3.11 Incremental full-text search of personal information. 38
3.12 Cross-lingual incremental search using an English-Chinese dictionary. 39
3.13 Incremental Search using a regular expression dictionary. 39

4.1 Traditional and modern writing environments. 42
4.2 Relation between human and computer. 43
4.3 Three types of information and their extents. 43
4.4 Dynamic abbreviation for “floccinaucinihilipilification”. 45
4.5 Dynamic abbreviation for “奈良先端科学技術大学院大学”. 46
4.6 Process of quick copy-and-paste. The user reuses information about a recently-published book. 47
4.7 Inputting “user interface” with Pen-based POBox. 48
4.8 Process of input prediction by directly reusing the previously written sentence. 49
4.9 Finding sample sentences for “morphological”. The user only types “morphol” at the moment. 51
4.10 Finding sample sentences for “thanks to”. 51
4.11 Checking an expression “Most people have difficulties in using computers” with our system. 53
4.12 Checking an expression “this are pens” 53

5.1 Illustration of the concept of mail agents. 66
5.2 Web pages created by the archiver agent. 67
5.3 Growth of QuickML users. 69
5.4 Number of posted messages. 70
5.5 Number of mailing lists. 71
5.6 Distribution of lifetime of mailing lists. 72
5.7 Distribution of number of members. 73

5.8 Correlation between size and lifetime of mailing lists. 73

List of Tables

2.1 BER compression for 32 bit positive integers. 12

2.2 Comparison between suffix array and inverted file. 15

2.3 Suffix array construction with a memory-poor machine. 20

2.4 Platforms used for the experiments on suffix array construction. . . . 22

3.1 Statistics in incremental search usage. 34

5.1 Comparison of basic functions with various mailing list systems. . . . 62

1 Introduction

We shall not cease from exploration and the end of all our exploring will be to arrive where we started and know the place for the first time.

— TS Eliot

As the Internet becomes popular, circulation of information increased rapidly because the Internet provides equal access to everyone who wishes to publish and share one's work. Many researchers have actively studied how to create and share information effectively. People can now publish their works on the Internet and share the work with others without asking a publisher or a media corporation. Creation and sharing of information can be seen as a continuous process of 1) how to find necessary information, 2) how to arrange the information, and 3) how to share the information with others. For example, if one wants to write a technical report on a new programming language and share the report with others, he or she follows the steps of:

1. Search the Internet for information on the programming language such as the official web site and introductory articles.
2. Arrange the collected information and finish the report while continually searching the Internet until sufficient information is collected.
3. Share and discuss the report with others by appropriate means such as email. If necessary, revise the report according to the discussion.

In this process, one has to follow these steps back and forth. It is thus important to regard the process as continuous. We consider that search and communication are

especially important activities in the process and effective assistance for the process can be achieved through the effective use of search and communication techniques.

1.1 Search and Communication

We argue that search and communication are especially important activities in the process of creation and sharing of information. Moreover, search and communication are two of the most popular activities on the Internet. In fact, people hardly find the information what they want without using *search* engine on WWW, and today's business cannot practically continue without *communication* using email. Jupiter Research[1] reported that email is the top online activity while search engine is second in 2002. Although the search and communication are the most popular activities, they are often considered separated process. We think that simultaneous study of both the search and communication is meaningful because they are the important activities on the creation and sharing of information as well as they are the most popular activities on the Internet.

1.2 Approach

The goal of this thesis is more effective creation and sharing of information through the effective use of search and communication techniques. We believe that circulation of information on the Internet has and will be increased substantially by supporting creation and sharing of information as a continuous process.

Incremental Search Method for Japanese Text

Although incremental search is used in a wide range of tasks including information retrieval and text editing, conventional incremental search method cannot handle Japanese texts effectively because Japanese text input requires an indirect input method called Kana-Kanji conversion. In this thesis, we propose a new incremental search method called Migemo to realize the incremental search for Japanese text. Migemo performs the incremental search by dynamically expanding the input pattern

[1] http://cyberatlas.internet.com/big_picture/geographics/article/0,,5911_1466661,00.html

into a compact regular expression which represents all the possible words that match the input pattern. We show how Migemo realize incremental search for Japanese text, which was hard to perform effectively before, through the concrete implementation and evaluation. Various applications are also presented.

Writing Assistance through Search Techniques

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 propose input acceleration systems and proofreading systems.

Instant Group Communication

A number of people are exchanging email messages everyday using mobile phones and PDAs as well as PCs. Email is useful not only for one-to-one communication but for group communication through mailing lists. However, conventional mailing lists are not as widely used as they should be, because creating and maintaining mailing lists is not an easy task. We propose a simple and powerful mailing list system called QuickML, with which people can easily create a mailing list and control the member account only by sending email messages. With QuickML, people can enjoy group communication at any place, at any time, and by anyone.

1.3 Organization of the Thesis

This thesis is organized as follows. Before going into the main topics, Chapter 2 briefly describes background search techniques and our implementations. We emphasize that a *suffix array* is especially useful for applied text processing using large corpora. In Chapter 3, we propose an incremental search method for Japanese text called *Migemo*

that improves usability of search operations, and show how Migemo effectively improves the search operations through the evaluation conducted with subjects who use Migemo regularly. In Chapter 4, we propose several methods for writing assistance through various search techniques, and show how the methods help writing substantially. In Chapter 5, we propose a group communication system called *QuickML* that allows users to share information with others easily, and show how QuickML effectively help users enjoy casual group communication through the public service involving thousands of users. Conclusion and future work are presented in Chapter 6.

2

Search Techniques

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

— Samuel Johnson

2.1 Introduction

Search is one of the most frequent activities in computing. In contrast to human, a computer is very good at memory for both accuracy and capacity. One can search hundreds of books and even the entire library in a second just by using a computer. In this chapter, we discuss various search techniques widely used in today's computing and propose our search systems called Namazu and Sary to investigate these search techniques and make good use of them for practical uses.

2.2 Sequential String Searching

The simplest string search algorithm is a brute-force algorithm. 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 with 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[29], *Boyer-Moore*[12] and *Shift-Or*[6] are linear algorithms in their worst-case. The first one is not so fast but theoretically valuable and others are fast and widely used. Practical comparison of performance among these algorithms is described in Chapter 8 of [7]. Aoe[5] collects classical papers in this field.

Even though efficient algorithms are employed, searching a large text file sequentially takes too long time. Instead of sequential searching, 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 2.3.1 and Section 2.3.2, respectively.

2.2.1 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 simplicity.

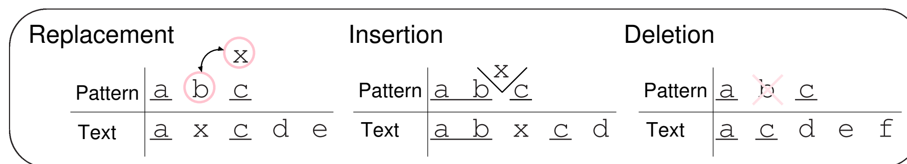


Figure 2.1: Three types of errors in a pattern.

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 an 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 a sample matrix of this algorithm. The time complexity of the algorithm is $O(mn)$ because $m{\cdot}n$ computations are required for completing the matrix.

text (t_j)

			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
pattern (p_i)	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: Dynamic programming for searching a text “failure” for a pattern “figure”. The lower right element indicates that the search is successful with two errors.

Non-deterministic Finite Automaton

Non-deterministic finite automata (NFA) is another approach to deal with errors. The algorithm constructs an NFA from a pattern and changes the states of the NFA as reading the text character by character. Figure 2.3 illustrates an NFA allowing two errors. The NFA accepts a text with at most two errors. Non-labeled transitions consume errors in the pattern.

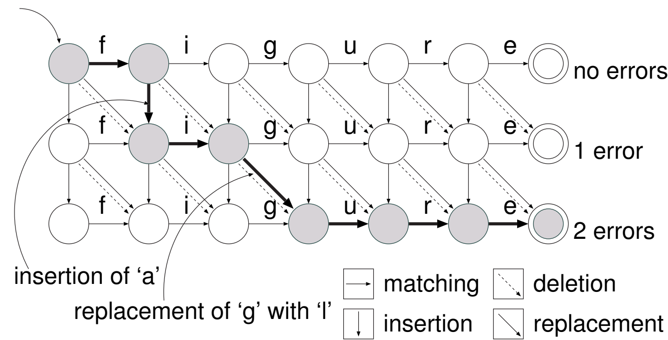


Figure 2.3: The NFA for a pattern “figure” with two errors. The shaded states and the bold lines illustrate the successful transition of reading a text “failure”.

An Advanced Algorithm

Wu and Manber[61] 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 fast bit operations of CPUs. The Unix command *agrep*[60] is their implementation of the algorithm.

Chapter 8 of [7] briefly summarizes this field and Hall[22] and Navarro[42] surveys this field comprehensively including a probabilistic similarity approach.

2.2.2 Regular Expression Searching

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

The difference between NFA and DFA is 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 a text. However, the transformation of NFA to DFA could be exponential to the length of a pattern in the worst case. Hopcroft

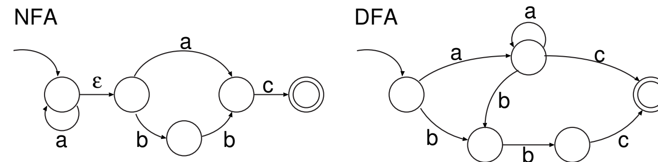


Figure 2.4: The non-deterministic and deterministic finite automata for a pattern `a*(a|bb)c`.

and Ullman[24] describes the automata theory and Friedl[17] discusses practical applications of regular expressions.

2.3 String Searching using Indices

Since sequential algorithms require time linear in the length of a text for searching, various data structures for improving the search performance have been proposed. In this section, we will discuss the data structures called an inverted file and a suffix array, which are popularly used in the field of information retrieval.

2.3.1 Inverted File

An inverted file is a data structure for fast string searching. The structure is originated from an index attached to the end of a book which allows readers to search for certain keywords quickly.

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

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.

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

Figure 2.5 illustrates the positions 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 2.6.

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

Figure 2.5: The sample text and the positions 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 2.6: 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 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 web pages containing certain keywords. Witten et al.[59] discussed and implemented a full-scale information retrieval system for millions of documents with an inverted file.

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 a 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 an ordinary sized main memory. Suppose that a text has 500,000 unique words[1] and the average length of the words is 10, the vocabulary consumes about 5 MB main memory only.

Compression of Inverted File

Since occurrences are stored in an ascending order, partial compression can be achieved by storing the positions in terms of their 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 [7].

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

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.

2.3.2 Suffix Array

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 suffixes of a text sorted in a 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 sorted suffixes in a lexicographical order.

Construction of Suffix Array

Figure 2.7 through 2.9 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 2.5. 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 2.7. Figure 2.8 shows the result of sorting. Suffixes are sorted in a lexicographical order and index points are reordered according to their corresponding suffixes.

Finally, the resulting index points become the suffix array for the sample text as shown in Figure 2.9.

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 2.7: The suffixes for the sample text and their index points.

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 2.8: The suffixes sorted in a lexicographical order. Each index point is reordered according to its corresponding suffix.

6	8	12	3	24	0	40	27	44	35	19

Figure 2.9: The suffix array for the sample text. Each entry corresponds to an index point shown in Figure 2.8.

Searching with Suffix Array

Searching of the sample text can be performed by binary search using the suffix array. Figure 2.10 shows the process of searching the sample text for "than." Numbered arrows show the order of the process.

	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 2.10: Searching the sample text for "than" by binary search using the suffix array. Numbered arrows show the order of the process.

The searching algorithm takes $O(P \log N)$ time where P is a length of a pattern and N is the length of a text. Manber and Myers[34] employ 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 construction as well as searching become complicated. For applications using short patterns, such as simple keyword searching, the factor of P can be negligible. Therefore, we do not employ the lcp information in this thesis.

Applications of Suffix Arrays

Applications of suffix arrays has been studied for years. In the field of natural language processing, it is important to process large-sized corpora. Yamashita[63] proposed a method of morphological analysis using a suffix array. This method directly uses POS (part of speech) –tagged corpora as example-base instead of constructing a dictionary from the corpora. Itoh[25] proposed a similar example-based method for word segmentation. Bentley[8] presented an application of suffix array for generating sentences with Markov chain algorithm. In the field of computational biology, Gusfield[21] investigated applications for exploring genome database.

2.3.3 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	easy	easy
Approximate searching	not easy	not 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 an inverted file is more efficient than a suffix array in terms of space.

- ***Construction time*** An inverted file can be constructed in $O(n)$ time while a suffix array takes $O(n^2 \log n)$ time, when usual sorting algorithm (e.g. quick sort)

is employed, where n is the length of a text. In other words, an inverted file can be constructed with a sequential read of the original text while suffix array construction requires random accesses over the original text $O(n^2 \log n)$ times.

- ***Incremental update*** When new texts are added, an inverted file can be updated efficiently by inserting newly added words 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 integer values of index points are randomly ordered. Recently, space-efficient data structures have actively been studied [20, 33, 48].

- ***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 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 pattern. However, errors crossing two or more words are hard to handle because a vocabulary is usually constructed as a set of single words. For example, approximate searching for “thebeatles” intended for “the beatles” is hard to perform even if a vocabulary list contains “the” and “beatles,” respectively. A suffix array can cope with approximate string searching by the method proposed by Yamashita[64], which applies an approximate tree traversal method.

- ***Word-oriented indexing*** Both inverted file and suffix array are 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 2.11 shows the result of morphological analysis for a Japanese sentence “真理は単純なものの中にひそむ” with ChaSen[39]. The leftmost column shows the segmented words and the others shows part of speech information for the words. The recall of searching using an inverted file highly depends on accuracy of word segmentation. In other words,

a poor segmentation leads to a poor recall. In contrast, a suffix array can cope with the text of such languages using a character-oriented indexing method.

- ***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 merging of occurrences takes a long time.

In summary, an invited file is usually suited for ordinary information retrieval task while a suffix array is also suited for information retrieval as well as applied text processing such as 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 2.11: Morphological analysis with ChaSen.

2.4 Implementations of Our Search Systems

We have implemented two search systems called *Namazu* and *Sary* using the data structures described in Section 2.3. Namazu is designed for searching vast quantities of documents, and Sary is designed for a single large document. In this section, we discuss the implementations of the systems and how they are applied in practice.

2.4.1 Namazu: A Full-Text Search Engine

Namazu[53] 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 and, the current implementation of Namazu has been developed by Namazu Project cooperatively. Namazu is a successful free software in term of popularity. In fact, Namazu version 2.x has been downloaded over a hundred thousand times and used at a large number of Web sites, including a rising open source desktop environment GNOME[2] at the present time. Figure 2.12 shows the mailing list search engine using Namazu at the GNOME website.

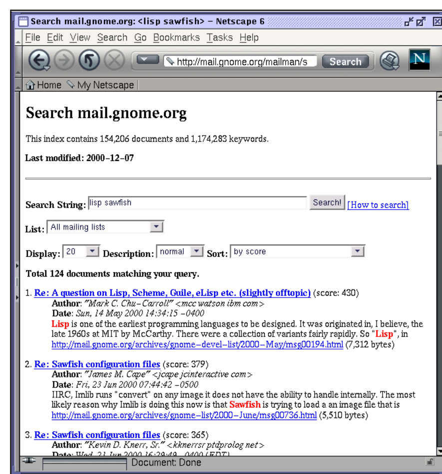


Figure 2.12: The mailing list search engine using Namazu at the GNOME website.

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

- ***Document Types*** Namazu can handle many types of documents including: *plain text*, *HTML*, *email*, *PDF*, *Microsoft Word*, *Microsoft Excel*, *Microsoft Powerpoint*, *UNIX manual*, and *TEX*. Some types of document requires external commands for text filtering.

[2]http://www.gnome.org/

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

- ***Handling of Japanese*** Namazu is capable of handling Japanese documents. Namazu provides interfaces for ChaSen and KAKASI[43] for word segmentation for Japanese texts.

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

Takabayashi[53] describes the system in detail. The author utilizes Namazu for his personal search engine. It is especially valuable for writing email for exchanging information with others.

2.4.2 Sary: A Suffix Array Library and Tools

Sary[54] is a suffix array library and tools. It provides fast full-text search facilities for a single large text on the order of hundreds of mega bytes such as newspaper corpora and genome databases. Sary is also freely available on the Internet as a free software.

Another suffix array library SUFARY[62] has been available since 1997. The difference between Sary and SUFARY is design philosophy. Sary is written in C but in object-oriented fashion. Sary aims for maintainability, extensibility, robustness, and usability. In fact, the library of Sary does not use global or static variables at all for making all functions reentrant so that the library can be used in multi-threaded programming.

Sorting Block by Block

Since suffix array construction requires $O((m+1)n)$ space where n is the length of a text and m is the size of an index point, processing a large text file in 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 character by character 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 that utilizes a hard disk as memory, suffix array construction in main memory is highly desirable because page faults caused by the virtual memory system reduces the construction performance. Accessing a hard disk is far slower than accessing main memory.

To reduce the memory requirement, Sary provides a method of memory-saving external sorting. The algorithm of sorting is described 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 character by character (i.e. the construction requires 175 MB memory) with a machine equipping with only 64 MB main memory. Table 2.3 shows that sorting with a 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 Sort	5359.28
Sort Block by Block	2118.12

Table 2.3: Suffix array construction with a memory-poor machine.

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 sorting block by block save memory, but also speeds up the suffix array construction. The performance of parallel sorting is discussed in Section 2.4.2.

Merging

Having completed sorting each block, the next stage is to merge of the sorted partial blocks. A simple algorithm is binary merging as shown in Figure 2.13. Let n be the number of total index points and m be 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.

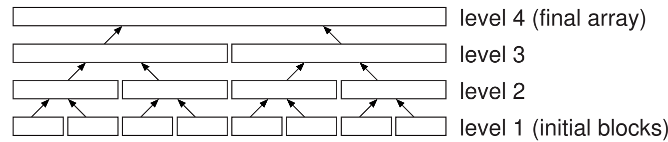


Figure 2.13: Merging sorted partial blocks in a binary fashion.

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 2.14.

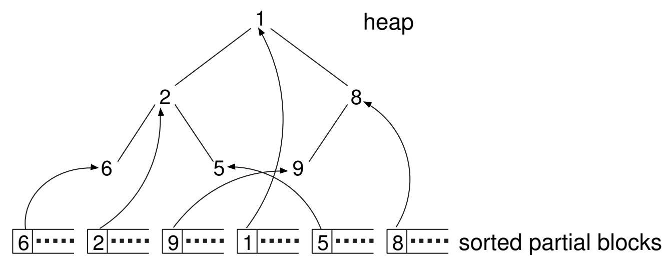


Figure 2.14: Merging sorted partial blocks with a heap structure.

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 $O(\log m)$ 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 our internal experiments. Theoretically, parallelized binary merging can be very efficient if plenty of processors are used. However, as discussed later, increasing the number of threads makes the performance poor even though a machine has plenty of CPUs.

Performance

Construction Time

We have conducted experiments for constructing a suffix array for a 80 MB text file at several block sizes. The experiments are conducted with two platforms as shown in Table 2.4. 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 limitation of the memory bus. Thus, the experiment is completed with two and four threads respectively to bring out their best performance.

Figure 2.15 shows 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. sorting the entire array at one shot in a single-thread) for comparison.

The result shows that multi-threaded sorting is faster than normal sorting for a large text file. It reduces at most 50 percent of time with the MIPS machine. Note that Sary employs *multikey quicksort*[9] as the sorting algorithm. Itoh[26, 27] proposed an even faster suffix array construction algorithm.

CPU	Memory	OS
Alpha 21164A 466 MHz $\times$ 4	2 GB	Digital UNIX V4.0F
MIPS R10000 195 MHz $\times$ 32	5 GB	IRIX 6.4

Table 2.4: 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 conducted 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.

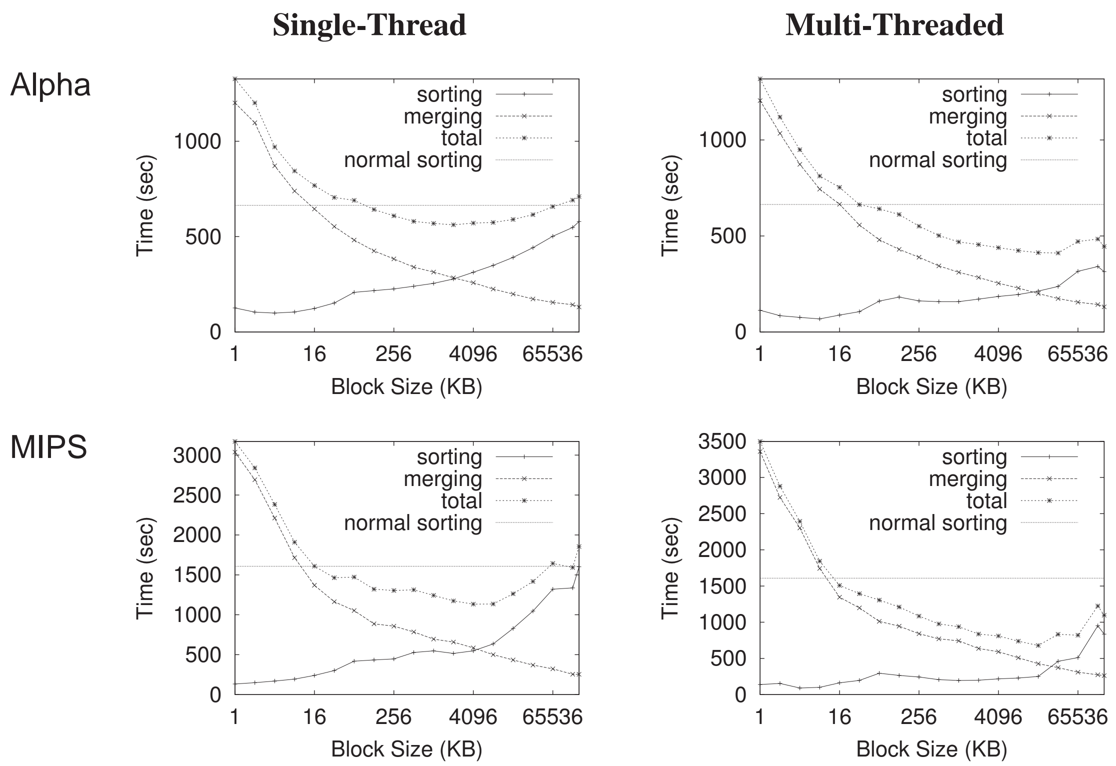


Figure 2.15: Suffix array construction with Alpha and MIPS machines. The right graphs show the results with multi-threading.

2.5 Summary

In this chapter, we discussed various search techniques from a brute-force method to a suffix array. It is worth noting that we have implemented two search systems called Namazu and Sary to investigate the search techniques and make good use of them for practical uses.

Although search is one of the most powerful functions in computing, it would not be so useful if user interfaces for searching is not well designed. We will discuss an improved a user interface called incremental search and propose a new incremental search method for Japanese text in the next chapter.

3
Incremental Search Method for Japanese Text

I have little respect for testing and evaluation in interface research. My argument, perhaps arrogant is that if you have to test something carefully to see the difference it makes, then it is not making enough of a difference in the first place.

— Nicholas Negroponte

3.1 Introduction

Incremental search is a method to proceed a search as a user types a keyword. In contrast to a common keyword search that requires a user to type the full spelling of a keyword as shown in Figure 3.1, an incremental search proceeds when a user type the first character of a keyword. An incremental search can be considered as one of the simplest form of Dynamic Query[51].

Text editors such as Emacs provide functions of incremental search for efficient editing of texts[52]. Incremental search is not only useful for searching for a keyword, but also useful for quick pointing (i.e. moving a cursor quickly to the position of a destination). Figure 3.2 shows the process of performing an incremental search for a word “scheme”.

In this example, a user typed only three keystrokes s, c, h to find “scheme”. In this way, a user can search for a longer word by typing only the first some characters.

Raskin[45] mentioned that incremental search is not only important for fast search-

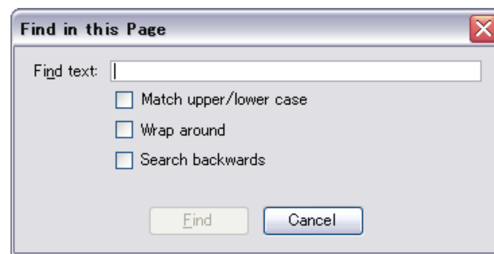


Figure 3.1: A common dialog for keyword search.

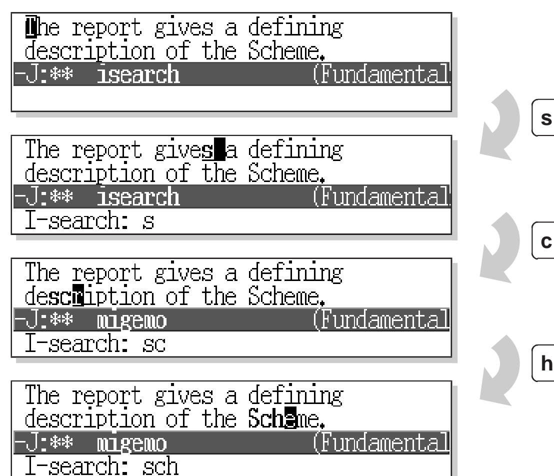


Figure 3.2: Incremental search for "scheme".

ing, but also for giving feedback to users as they type a keyword. Raskin designed a machine of a word processor equipped with special keys only for incremental searches.

Incremental search is one of the most powerful operations for editing of texts. However, incremental search is not directly applicable to Japanese, where keyboard characters do not directly correspond to text characters. A user have to perform *Kana-Kanji* conversion to input a Japenese word[1].

In incremental search, it is very important to proceed a search as a user types a keyword. However, Kana-Kanji conversion makes it impossible to proceed an incremental search smoothly because Kana-Kanji requires a user to do the following three steps.

[1]Some direct input methods for Japanese such as t-code[44] are available but they are seldom used because remembering special keystrokes for thousands of Kanji characters is too hard for casual users.

1. Type the pronunciation of a word using an ASCII keyboard. The pronunciation is automatically converted into a Kana text that represents the pronunciation of a Japanese word.

2. Convert the Kana text into a set of Kanji words by a Kana-Kanji converter. One pronunciation usually corresponds to more than one Kanji characters. For example, "機械 (machine)", "機会 (opportunity)", and "奇怪 (strange)" all have the same pronunciation "きかい (kikai)".

3. Select the desired Kanji word from the set of candidate words.

If users want to find "奇怪" with a conventional incremental search method, they have to type more than ten keys[2]. Figure 3.3 shows the process of selecting "奇怪" as a search keyword. A search with Kana-Kanji conversion in this way is thus not incremental at all.

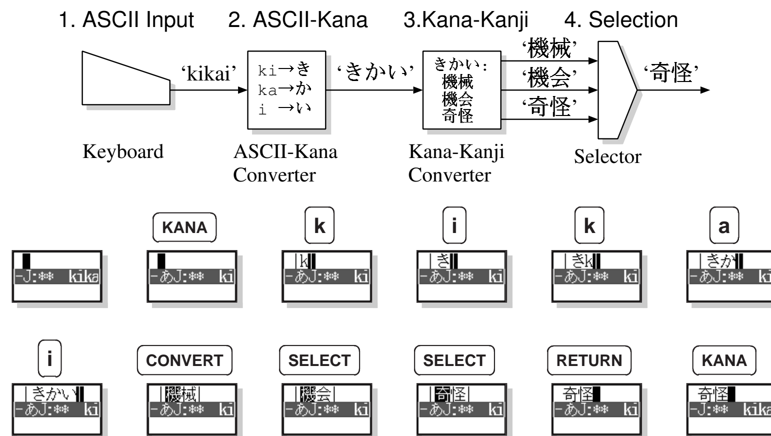


Figure 3.3: The process of selecting "奇怪 (strange)" as a search keyword.

[2] Using Emacs, the users have to type ^S KANA k i k a i CONVERT SELECT SELECT RETURN KANA: ^S starts an incremental search, KANA starts and ends Kana-Kanji conversion, CONVERT converts a Kana text to a set of Kanji words, SELECT selects a Kanji word from the candidates of the Kanji words, and RETURN finishes the selection process of Kanji words.

Figure 3.4: Incremental search for 「奇怪 (kikai)」 using Migemo.

3.2 Migemo

We propose a new incremental search method called Migemo [55, 56], which solves the problem described in Section 3.1. Figure 3.4 shows the process of performing an incremental search for a Japanese word "奇怪" with the Migemo for Emacs.

In this example, the user types only four keys ^S k i k to find "奇怪", which is much easier than typing twelve keys in the previous example. Unlike conversion-based search, the search with Migemo is truly incremental and dynamic, just like an incremental search for ASCII documents.

3.2.1 Method

Migemo skips Kana-Kanji conversion by dynamically expanding an input pattern into a compact regular expression which represents all the possible words that match the input pattern. Figure 3.5 shows the generated regular expressions for "k", "ki", and "kik" respectively.

Although the actual search is performed by the expanded regular expressions, a user does not have to care about the expansion process because the expansion is performed silently in the background.

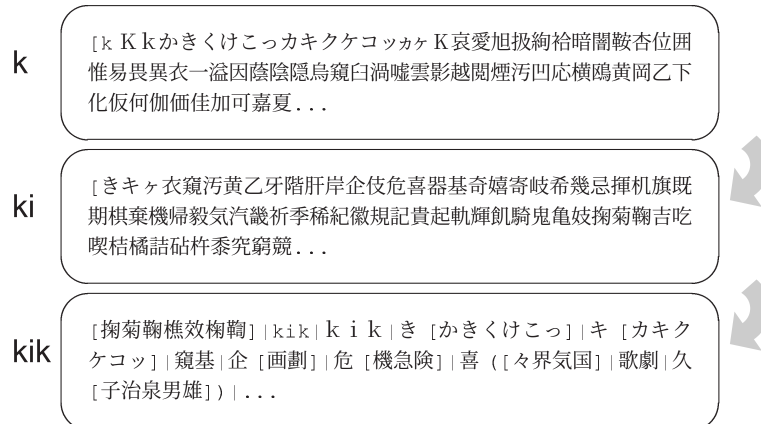


Figure 3.5: Generated regular expressions for “k”, “ki”, and “kik”.

3.2.2 Generation of Regular Expressions

Migemo generates a regular expression using ASCII-Kana conversion rules and a dictionary of Japanese words.

The first regular expression in Figure 3.5 includes the candidates generated by the ASCII-Kana conversion rules to cover all the Japanese Hiragana and Katakana characters beginning with “k”:

k ⇒ か | き | く | け | こ | っ | カ | キ | ク | ケ | コ | ッ | ｶ | ｹ

To generate a regular expression for a pattern “kik”, Migemo first expands “kik” to “きか (kika)”, “きき (kiki)”, “きく (kiku)”, “きけ (kike)”, “きこ (kiko)”, “きっ (ki + doubled consonant)” using the ASCII-Kana conversion rules and extract words beginning with these prefixes in a dictionary. The extraction of Kanji words is performed by looking up the words in a Japanese word dictionary. Figure 3.6 shows an excerpt from our dictionary.

The simplest method to generate a regular expression from a set of words is to concatenate the words using *or* operator “|”. However, this naive method often generates huge and redundant regular expressions when it handles a large set of words. In fact, our dictionary contains thirty hundreds of words that begin with “k” so that the naive method result a regular expression 186 kilo bytes long in length. We can see that the regular expression by the naive method contains words having the same prefix as

Pronunciation		Representation
· きかい (kikai)	→	機械 機会 奇怪 器械 貴会 気塊 喜界
· きかいあぶら (kikaiabura)	→	機械油
· きかいぶひん (kikaibuhin)	→	機械部品
· きかいちのう (kikaichinou)	→	機械知能
· きかいがっかい (kikaigakkai)	→	機械学会
· きかいがく (kikaikagku)	→	機械学
· きかいがくしゅう (kikaigakushuu)	→	機械学習

Figure 3.6: The dictionary for generating regular expressions (excerpt).

The original regular expression

mour | ｍｏｕｒ | もうっ | モウッ | もうら | モウラ | 網羅 | 網羅性 | 網羅的 | 網羅率 | もうり | モウリ | 毛利 | 魍魎 | もうる | モウル | もうれ | モウレ | 猛烈 | 猛烈巨人 | 猛烈不況 | 猛練習 | 猛連荘 | もうろ | モウロ | 朦朧 | 耄碌

The optimized regular expression

mour | ｍｏｕｒ | もう［っらりるれろ］| モウ［ッラリルレロ］| 毛利 | 猛（烈 | 練習 | 連荘）| 網羅 | 朦朧 | 耄碌 | 魍魎

Figure 3.7: Optimization of the regular expression for “mour”.

follows.

機械油 | 機械部品 | 機械知能 | 機械学会 | 機械学 | 機械学習

These redundant patterns causes a problem of performance because it requires an NFA (non-deterministic finite automaton) engine to do backtracks repeatedly[17]. In addition, some NFA engines rejects such a large regular expression.

To solve the problem, Migemo optimizes a regular expression by unifying the redundant words. Figure 3.7 shows a process of the optimization for an input pattern “mour”.

In this example, words such as “網羅 (moura)”, “網羅性 (mourasei)”, and “網羅的 (mourateki)” are unified to the shortest word “網羅”. Similarly, words such as “も

うっ(mou + doubled consonant)", "もうら (moura)", and "もうり (mouri)" are unified to "もう［っらりるれろ］" using a character class. Moreover, words such as "猛烈 (mouretsu)", "猛練習 (mourenshuu)"," 猛連荘 (mourensou)", that begins with "猛 (mou)" are unified to "猛（烈｜練習｜連荘）" using parenthesises. The optimization algorithm is shown as follows:

1. Group a set of strings having a same prefix.
   e.g.: "網羅｜網羅性｜網羅的｜朦朧｜..."
   ⇒ "（網羅｜網羅性｜網羅的)｜朦朧｜..."
2. Group a set of substrings subsequent to the common prefix.
   e.g.: "網羅｜網羅性｜網羅的" ⇒ "網羅（｜性｜的）"
   (a) If an empty string is found in a group, remote the group.
       e.g.: "網羅（｜性｜的）" ⇒ "網羅"
   (b) If the length of all strings in a group is one, group them as a character class.
       e.g.: "も（あ｜い｜う｜え｜お）"
       ⇒ "も［あいうえお］"
   (c) Apply the same algorithm in a set of strings recursively.
       e.g.: "も（あい｜あう｜い）"
       ⇒ "も (あ［いう］｜い)"
3. Terminate if a set of words having a same prefix is no longer found.

In this way, the optimization reduces a length of a regular expression. For example, if the naive method is used, the generated regular expression for "k" is 186,446 bytes while our optimization reduces it to 4,994 bytes.

In addition, the optimization method improves a performance of a search using NFA engine because it reduces redundant pattern matches. For example, a search of a one-mega-bytes text file using the optimized regular expression in Figure 3.7 takes only 1.1 seconds while the original regular expression takes 2.5 seconds, using GNU Emacs 21.2 and a PC equipped with a CPU of Pentium III 1 GHz.

The performance is a very important factor in incremental search not to feel stress. To respond to user's keystrokes in real time, Migemo uses pre-compiled regular expressions for short input patterns such as "a", "k", and "s" instead of dynamically

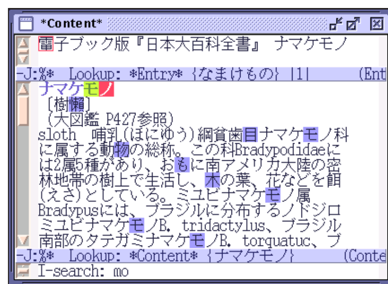


Figure 3.8: Migemo for Emacs.

generating a regular expression using a dictionary. Although our implementation of Migemo needs 3.65 seconds to generate a regular expression for "k" using the PC described above, practical performance is achieved through the use of pre-compiled regular expressions. In addition, regular expressions for longer patterns such as "kik" can be generated within 0.1 seconds in real time so that Migemo responds to user's keystrokes quickly.

A time taken by a search using the generated regular expressions differ whether the search is succeeded or not. On the one hand, in a text editor, an incremental search suspends at the nearest matched position from the original position of a cursor. On the other hand, if no matched positions are found, a search proceeds to the end of a text, thus the time taken by the search is proportional to the length of the text. As discussed above, our implementation can search a one-mega-bytes text within one second so that searching of usual text files can be performed fast enough.

3.2.3 Implementations

We implement Migemo as an extension to the incremental search function of Emacs (Figure 3.8). Users can do a Japanese incremental search in the same way of a usual incremental search.

Migemo is freely available on the Internet as a free software and used by many users. Currently, Migemo has been ported to a text-based web browser called *w3m*, a text editor *VIM*, and a text editor *xyzzy* (Figure 3.9). These ports of Migemo support the usefulness of our method.

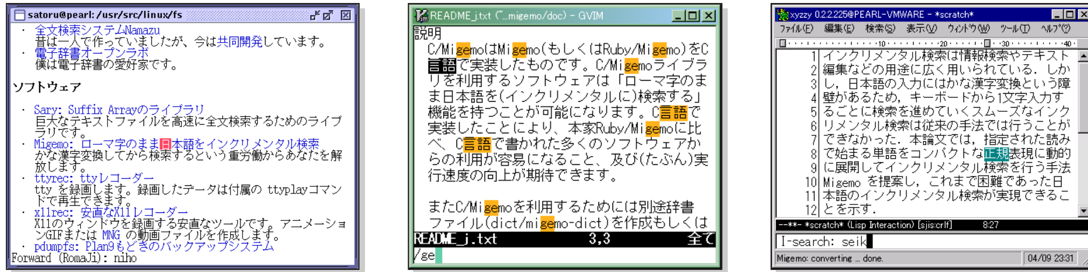


Figure 3.9: Migemo implementations: w3m, VIM, and xyzzy.

3.3 Evaluation

We conduct an experiment to measure effectiveness of Migemo quantitatively. We collect a daily log of usage from subjects who use Migemo regularly and analyze the log.

The log includes usage data such as patterns used in searches, strings found by searches, time taken for searches. In addition, the log includes editing modes of Emacs editor to investigate what kind of document is edited when Migemo is used. The editing mode changes according to a type of a document edited by users. For example, if a user is editing a C program, the editing mode would be c-mode, and if a user is editing a TEXdocument, the editing mode would be latex-mode.

Table 3.1 shows usage of incremental searches collected by English, Japanese, and total separately. The ratio of incremental searches in Japanese over total incremental searches is presented as a percentage.

The table illustrates the fact that user A does incremental searches in English an average of 126.71 times per day, incremental searches in Japanese an average of 55.70 per day, and incremental searches in total an average of 182.41 times per day. It is notable that incremental searches in Japanese amounts to nearly one third (usage ratio 30.05%) for the user A. Similarly, user E does incremental searches in Japanese many times (usage ratio 30.78%). The log data reveals that these users commonly do incremental searches in Japanese while they are editing TEXdocuments written in Japanese.

However, user B, C, D, and F only do some incremental searches in Japanese. We analyzed their log data to investigate editing modes they used and find that they

User	Incremental Searches in English			Incremental Searches in Japanese				Incremental Searches (Total)		
	Av. # of times	Av. keystrokes	Av. time (sec)	Av. # of times	Av. keystrokes	Av. time	Ratio %	Av. # of times	Av. keystrokes	Av. time
A	126.71	5.32	2.95	55.70	5.14	2.61	30.05	182.41	5.25	2.83
B	72.04	4.50	2.00	3.51	3.92	2.11	5.65	75.55	4.46	2.01
C	105.02	7.79	4.59	4.00	3.75	5.63	3.67	109.02	7.52	4.66
D	406.93	8.66	3.10	23.14	5.10	1.99	5.38	430.07	8.40	3.02
E	113.26	6.22	2.84	50.31	4.75	2.27	30.78	163.58	5.68	2.63
F	111.97	4.39	4.14	4.61	4.50	2.67	3.95	116.58	4.40	4.06

Table 3.1: Statistics in incremental search usage.

Note: Av. = Average. # = number. Average number of times is calculated by per day basis.

were doing incremental searches while they were doing programming, particularly the user B did Unix shell programming (shell-mode), the user C and D did Emacs Lisp programming (emacs-lisp-mode), and the user F did C programming (c-mode). In these situations of programming, Japanese text is not much included. Therefore, incremental searches in Japanese is not quite useful.

These results show that users editing a Japanese documents do incremental searches in Japanese nearly one third of the time and the incremental search method realized by Migemo helps users effectively.

3.4 Discussion

We design and implement Migemo with a high priority on practical uses so that we can use it for our everyday tasks. In this section, we discuss the trade-offs and decisions in the design made and implementation of Migemo.

3.4.1 Other Approaches

While Migemo performs incremental search in Japanese by expanding an input pattern into a regular expression, similar search is also possible if Japanese texts are converted to ASCII texts before the search.

The simplest method is to convert the whole Japanese text into an internal ASCII text at every search. Incremental search can be realized by searching the internal ASCII

text. For example, a Japanese text “テキストと漢字” is converted into “tekisuto to kanji”. However, since the conversion requires time linear in the length of the text, converting the large text at every search is thus not practical.

There are other methods such as converting the whole text into an internal ASCII text when a file is read, or converting a partial text dynamically into the internal ASCII text as an incremental search proceeds. However, at any rate, the problem of inaccurate Kanji-ASCII conversion remains. For Japanese text, Kanji-ASCII conversion is not always accurate because of ambiguity of Kanji expressions. For example, if a word “博士 (doctor)” appears solely, even human cannot distinguish whether it is pronounced as “hakushi” and “hakase”.

Migemo has an advantage to the problem because Migemo can search for “博士” by “hakushi” or “hakase”. Additionally, our method that expands an input pattern into a regular expression relatively easy to implement.

3.4.2 Handling of Homonyms

Migemo might find inappropriate words accidentally because of homonyms. For example, a user might find “機械 (machine)” instead of “奇怪 (strange)” when the user want to find “奇怪”, caused by the same pronunciation “kikai”.

In our experience of using Migemo for over years, this kind of situation rarely occurs. Even though the unintended matching occurs, a user can simply ignore it by proceeding to the next matching position. We investigate a paper on Migemo[56], that is written in Japanese to see how many homonyms appears in the paper using a morphological analyzer. The Figure 3.10 shows the all homonyms appeared in the paper of eight-pages. Out of the homonyms, “器械”, “奇怪”, “機会”, “機械”, “串”, “櫛”, “烈”, “葉”, “猛”, and “機” are used for explaining the expansion method of a pattern into a regular expression, and we can conclude that there are only a few homonyms in the regular text of the paper.

The characteristic that Migemo accidentally find inappropriate homonyms could be an advantage of tolerance to inconsistency of word representations. For example, a user can find “こんにちは (hello)” and “今日は (hello)” by the same pattern “konnnichiha”. Similarly, our dictionary includes similar representations like “インターフェース”, “インターフェイス”, “インタフェース”, and “インタフェイス” for the single pattern “interface” for allowing variants in the representations in a text so that a user can find

Pronunciation		Representation
· い (i)	→	い 位
· いち (ichi)	→	位置 一
· か (ka)	→	か 化
· き (ki)	→	き 機
· きかい (kikai)	→	器械 奇怪 機会 機械
· くし (kushi)	→	串 櫛
· ない (nai)	→	ない 内
· は (ha)	→	は 葉
· もう (mou)	→	もう 猛
· よう (you)	→	よう 用 要
· れつ (retsu)	→	列 烈

Figure 3.10: Homonyms occurred in the text of a Japanese paper on Migemo.

them with the same pattern "interface".

3.4.3 Handling of Multi-Segment Phrases

Japanese texts are written without spaces. A phrase consisting of three words "現在 (current)", "の (of)", and "実装 (implementation)" is represented as "現在の実装 (current implementation)" without spaces. The early version of Migemo could not search for such a multi-segment phrase.

We solve the problem by using the method proposed by a Kana-Kanji conversion system called SKK[1]. Using the method, a user has to mark the beginning of each segment by capital letters. For example, "現在の実装 (genzainojissou)" can be found by just typing G e n z a i N o J i s s o u[3]. Migemo generates regular expressions for each segment and concatenate them to search.

We considered another solution that generates a regular expression covering all combinations of multi-segment phrases. However, we rejected the approach because the number of combinations explode very fast. If we employ grammatical rules to reduce the number of combinations, the generation of a regular expression becomes

[3]It could be a small letter g because the beginning of the first segment is obvious.

very complicated. For example, the regular expression for “karehahakushidesu (he is a doctor)” becomes this complicated pattern “（彼は（博士|白［紙詩］|薄志）|枯（れ葉|葉）は［串櫛］）です” and it could become more complicated as a phrase become longer. For this reason, we concluded that fast generation of a regular expression by the approach is thus not practical.

3.4.4 Use of a Dictionary

Since Migemo uses a dictionary to generate regular expressions, Migemo cannot find words that are not defined in the dictionary, just like a Kana-Kanji conversion system cannot convert a Kana string to a word which is not defined in the dictionary.

Because of this, Migemo could fail to find a word even if the word appears in a text. Since Migemo itself cannot distinguish whether the word is not found actually in the text or the word is not just defined in the dictionary, users have to distinguish it for themselves.

The judgment is not difficult when a user searches for a common word obviously defined in the dictionary. However, the judgment could be difficult when the user searches for an unusual word such as a proper noun. In that case, the user has to search for the word by a conventional method to be sure that the word is not found actually or else. This problem, called unknown word processing, that a word not defined in a dictionary is not searchable, can be reduced by enriching the vocabulary of the dictionary Migemo uses, just like enriching the dictionary of a Kana-Kanji conversion system to improve the accuracy of the conversion.

The method for multi-segment phrases discussed above can be used for searching a word not defined in a dictionary. For example, a person’s name not defined in a dictionary “契沖 (hisaoki)” can be searched by a pattern “KeiChuu” because “契沖” can be decomposed as “契 (Kei)” + “沖 (Chuu)”.

3.5 Applications

In this section, we describe two applications of Migemo. One is a method for sharing a dictionary with an input system and another is a method for language-independent incremental search.

3.5.1 Sharing a Dictionary with an Input System

Incremental search is particularly useful for personal information management (PIM) systems. Masui[37] applied Migemo to a PIM system called *Q-Pocket* for full-text retrieval. It is much easier to search Japanese texts from direct Graffiti input than selecting a Japanese keyword before performing a search. Here, Migemo uses a Japanese dictionary used in a mobile text input system called *POBox*[35]. Using the same dictionary both for text input and for text search has an advantage of consistent text handling. If a word "☆" is defined to have a pronunciation "star" in the dictionary, a user can type [s] [t] [a] [r] to enter "☆" using an Kana-Kanji conversion method. Documents including "☆" can be retrieved by also typing [s] [t] [a] [r]. To realize fast full-text searching, Q-Pocket employs Aho-Corasick[4] method for handling *or*-connected regular expressions. Figure 3.11 shows how incremental full-text search of memos and schedule data is performed with Q-Pocket.

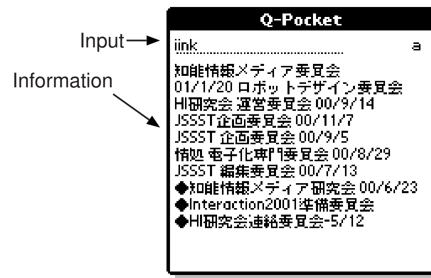


Figure 3.11: Incremental full-text search of personal information.

3.5.2 Language-Independent Incremental Search

Applications of Migemo are not limited to handling documents written in Japanese, but it is useful for handling documents written in any languages. For example, if a user have an abbreviation dictionary that includes entries such as:

- jfk → John F. Kennedy
- jfk → John Fitzgerald Kennedy

they can type [j] [f] [k] to find both "John F. Kennedy" and "John Fitzgerald Kennedy". If a user of an HTML editor can use a dictionary that includes entries such as:

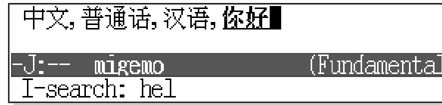


Figure 3.12: Cross-lingual incremental search using an English-Chinese dictionary.

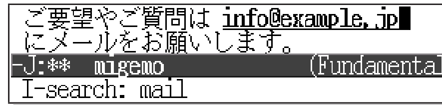


Figure 3.13: Incremental Search using a regular expression dictionary.

- header → <h1>
- header → <h2>
- header → <h3>

he or she can incrementally search for all the header tags by typing h e a d e r.

Figure 3.12 shows how to search " 你好 " ("hello" in Chinese) by typing h e l l o using an English-Chinese dictionary including an entry:

- hello → 你好

A user can also add regular expressions as well as usual words to a dictionary. For example, if a user has a dictionary including a regular expression that matches mail addresses such as

- mail → [-0-9a-zA-Z_.]+@[-0-9a-zA-Z_.]+

the user can search for mail addresses just by typing m a i l (Figure 3.13). In this way, a user can also search for URLs just by typing u r l with a regular expression that matches URLs.

3.6 Summary

We propose a new incremental search method called Migemo for Japanese text and evaluated how Migemo is effectively used. Migemo can be used in various situations

including incremental searches in text editors and text retrieval in PIM systems. We are aiming to popularize Migemo much further so that any applications have the capability of incremental search for Japanese text.

In the next section, we will discuss how the search techniques discussed in Chapter 2 and the incremental search method described in this chapter can be used together for writing assistance.

4

Writing Assistance through Search Techniques

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

4.1 Introduction

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 has been decreasing rapidly. Figure 4.1 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 and correct spellings and even grammars semi-automatically with computers.

However, there are still many scopes to improve the writing environment. In this

chapter, we propose several methods for realizing a better writing environment.

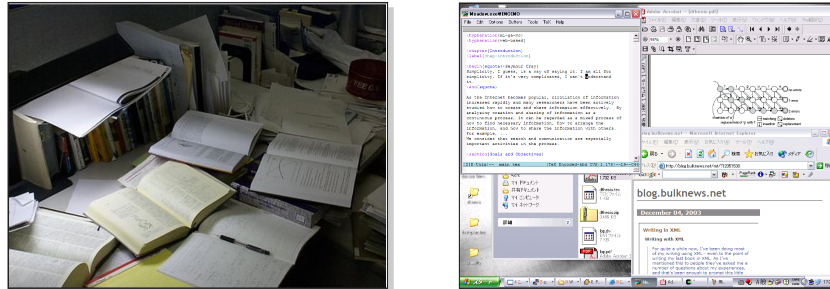


Figure 4.1: Traditional and modern writing environments.

4.2 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 possibly restricted by their tools (i.e. computers). In other words, computers interfere with train of thoughts.

Figure 4.2 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.

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 functions 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 4.3 depicts three types of information and their extents.

We frequently search external and "available somewhere" information by several ways for various kind of writing. These types of information are especially neces-

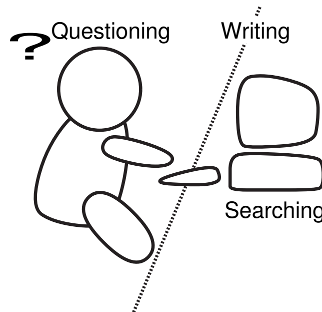


Figure 4.2: Relation between human and computer.

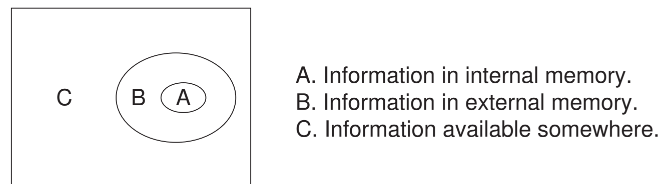


Figure 4.3: Three types of information and their extents.

sary 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.
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 could 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.

4.2.1 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 chapter, not only do we present several methods for writing assistance employing various search techniques, but also we propose the methodology "Writing is Searching." We believe that the integration of writing and searching opens a way for a better writing environment.

4.3 Input Acceleration

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 section, we discuss several methods for input acceleration.

4.3.1 Dynamic Abbreviation

Dynamic abbreviation is a technique for input acceleration. It saves keystrokes for inputting previously-input words. For example, if a user wants to input "floccinaucinihilipilification" for the second time, dynamic abbreviation allows the user to input

“floccinaucinihilipilification” with short keystrokes. Figure 4.4 illustrates a process of dynamic abbreviation with Emacs editor. The user types only two keystrokes: f META-/ where META-/ key triggers the dynamic abbreviation function. In this case, it saves as many as 27 keystrokes.

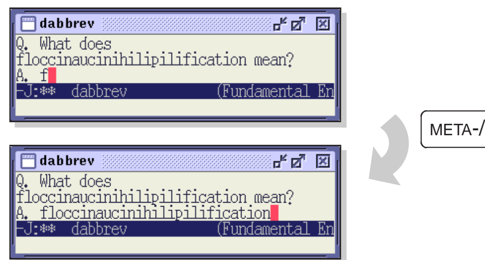


Figure 4.4: Dynamic abbreviation for “floccinaucinihilipilification”.

The mechanism of the dynamic abbreviation is fairly simple. As in the example, the dynamic abbreviation first searches for a word which begins with “f” backward and then insert the word into the point where the cursor stays.

A problem arises when performing dynamic abbreviation for Japanese words because of Kana-Kanji conversion. For example, performing the dynamic abbreviation for “奈良先端科学技術大学院大学 (nara sentan kagaku gijutu daigakuin daigaku)” requires the user to type many keystrokes for inputting the first single word “奈良 (nara)”: KANA n a r a CONVERT KANA and then type META-/ to expand “奈良” to the whole phrase “奈良先端科学技術大学院大学”.

To solve this problem, we propose a dynamic abbreviation method for Japanese words, that bypassed a Kana-Kanji conversion. Using an incremental search method for Japanese text discussed in Chapter 3. Figure 4.5 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 a word which begins with “na” backward. Then, it insert the word consisting of all Kanji characters.

However, our method is imperfect because it cannot distinguish the end of a word to be expanded accurately. For example, if a user want to expand “nara” to “奈良先端大” while the user is editing a text including “奈良先端大食堂にて (nara sentan dai syokudou nite)”, the method inappropriately expand “nara” to “奈良先端大食堂”

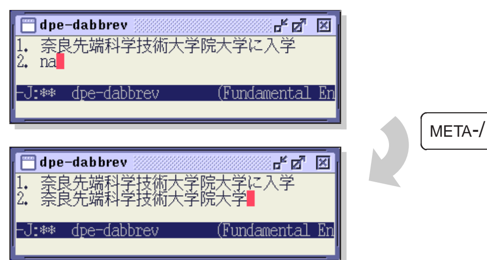


Figure 4.5: Dynamic abbreviation for "奈良先端科学技術大学院大学".

because these characters are all Kanji and regarded as a single word naively by the method, that should be regarded as a phrase of two words "奈良先端大" and "食堂".

To solve the problem, called word boundary ambiguity, Komatsu[31] extended our method and proposed a dynamic abbreviation method called *Nanashiki* by employing a morphological analysis for distinguishing the end of a word. Using Nanashiki, a user can expand "nara" to "奈良先端大" appropriately in the above situation because it accurately distinguish the boundary of Kanji words using a morphological analyzer.

4.3.2 Quick Copy-and-Paste

In the field of software engineering, importance of reusing software components 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 a new document and copy-and-paste the previously-written sentence in their current work instead of composing a new sentence from scratch. Masui[36] 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 a number of documents.
2. Selecting an appropriate sentences, phrases, or expressions.
3. Copy-and-Paste.

This process could make people exhausted. We propose a method to reduce obstacles of reusing information. Our method integrate the three steps into a single step by

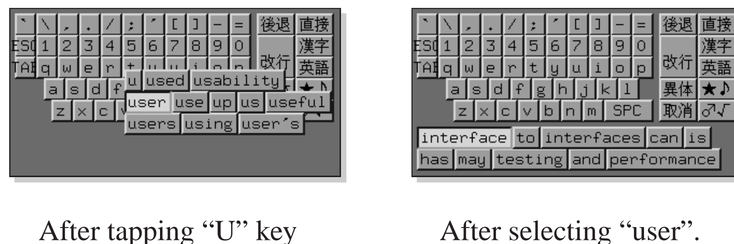
employing a suffix array and a method of regular expression generation discussed in Chapter 3. The suffix array is used for fast full-text searching and the regular expression generation is used for text searching without doing Kana-Kanji conversion.

Figure 4.6 illustrates a typical process of reusing information with our system. A user is wishing 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 about the book 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.

Figure 4.6: Process of quick copy-and-paste. The user reuses information about a recently-published book.

4.3.3 Input Prediction

Masui[35] 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 4.7 shows 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 4.7 shows that “interface” is the first candidate because “interface” is likely to follow the word “user”.



After tapping “U” key After selecting “user”.

Figure 4.7: Inputting “user interface” with Pen-based POBox.

Although input prediction can be a very effective way to input words, prediction of the next word is not an easy task. One approach to perform the task is to construct a dictionary for prediction with an n-gram model. However, an n-gram model limits its prediction capability to *n* words. We propose a method for input prediction by searching previously-written documents as an example-base and directly reusing them instead of constructing a 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 4.8 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.

The implementation of the system is very similar to the system for quick copy-and-paste, discussed in Section 4.3.2. Since our system is in the early stage of development, the current implementation uses only raw texts. Linguistic knowledge like part-of-

Figure 4.8: Process of input prediction by directly reusing the previously written sentence.

speech information should also be employed for more accurate prediction.

4.4 Proofreading

Writing an error-free document is desirable for expressing ideas without being misunderstood by readers. However, 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.

4.4.1 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 July 2003, Zakon[65] have reported that the Web has grown to more than forty million web servers.

Furthermore, papers in various fields are also available and it is possible to utilize the papers as sample sentences. One such source freely available is e-Print archive[1]. This website offers a large quantity of papers in a variety of fields.

Consequently, utilizing such a large quantity 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 hundreds mega bytes of sample sentences. To reduce the search time, we should employ certain data structures. Since finding sample sentences requires the efficient phrase searching, we chose a suffix array instead of an inverted file for searching of a large quantity of papers.

To utilize a number of papers as sample sentences with a suffix array, one should concatenate the papers to a single large text file and construct a suffix array for the concatenated file. Although concatenating the papers loses identities of documents, it is not a problem because information about which document contains a sentence is not

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

so important in the task of finding sample sentences. Having the constructed suffix array, searching of the papers can be performed very fast. Figure 4.9 shows a snapshot of our system within the Emacs editor. Our system dynamically finds sample sentences as a user types search strings just like an incremental search. The response time is fast enough thanks to the suffix array. Phrases can also be searched similarly as shown in Figure 4.10.

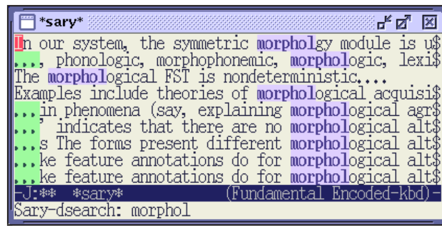


Figure 4.9: Finding sample sentences for “morphological”. The user only types “morphol” at the moment.

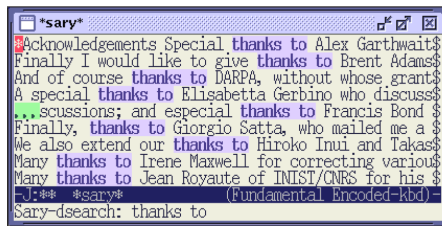


Figure 4.10: Finding sample sentences for “thanks to”.

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. We should incorporate methods of example-base natural language processing such as CTM[50] to solve the problem.

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

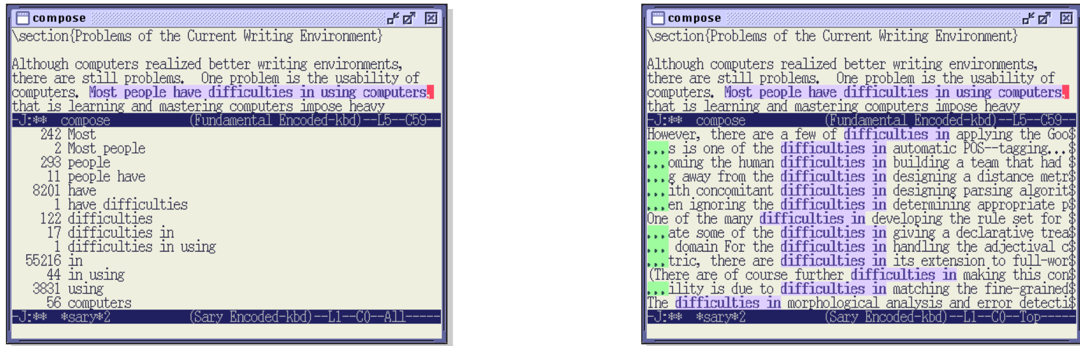
scored" and "are scored" simultaneously by simply typing "*be* scored". Similarly, different tense and countability should be dealt. These problem can be solved through the use of regular expression searches. For example, it is possible to expand "be" to the regular expression pattern "(is|am|are|was|were)" using rules and a dictionary.

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.

4.4.2 Checking Expressions

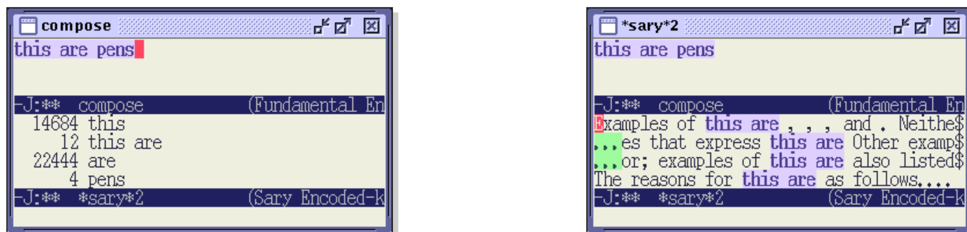
Non-native writers often worry about which expression is better in writing. One way to check whether an expression is natural or not is to consult sample sentences as discussed in Section 4.4.1. We present a more efficient way to check the expressions. Our method employs documents written by native writers as example-base and check expressions by comparing with the example-base. Figure 4.11 illustrates checking of an expression "Most people have difficulties in using computers" with our system. The left window shows the frequencies of phrases in the expression. With such information, a 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 4.4.1. Checking an expression and finding sample sentences are seamlessly integrated. We think that this method could be extended to a spell checker if a plenty of documents with no typos is provided.

While our method helps substantially, 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 possibly misbelieve 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 4.12 illustrates the situation. To solve this problem, advanced linguistic knowledge like grammatical construction is required.



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

Figure 4.11: Checking an expression "Most people have difficulties in using computers" with our system.



The system shows the frequency of "this are" is 12. "this are" in the corpus are used in appropriate ways.

Figure 4.12: Checking an expression "this are pens"

4.5 Summary

In this chapter, we present various applications of search techniques for writing assistance in terms of our methodology “Writing is Searching”. We propose three methods for input acceleration including dynamic abbreviation for Japanese words, quick copy-and-paste, and input prediction. We also propose two methods for proofreading including finding sample sentences and checking expressions.

While we believe that these techniques help people to write technical documents effectively, there are many scopes of improvements. First, searching of a raw text limits applications of writing assistance. There are problems which cannot be solved by 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 to use a suffix array for searching static information only and use other search techniques for dynamic information. Finally, more effective assistance should be achieved. We are considering employing techniques of query-free information retrieval[23] that autonomously constructs queries and presents information relevant to a user, and techniques of just-in-time information retrieval that presents information relevant[46, 47] to a user based on a user’s local context.

While we study on writing assistance, we also have been actively interacted with other researchers . One effort made by a collaboration with other researchers is a predictive text input system using a large number of self-written documents[30]. We hope that we can continue the research in a collaboration with a broader range of researchers and achieve significant results.

In this section, we discussed writing assistance and creation of information is eased by the assistance. In the next chapter, we will discuss how to easily communicate with others for sharing the created information.

5
Instant Group Communication

Simplicity, I guess, is a way of saying it. I am all for simplicity. If it's very complicated, I can't understand it.

— Seymour Cray

5.1 Introduction

As the Internet becomes popular, email is indispensable as a daily communication medium[3, 16]. Email is useful not only for one-to-one communication but group communication through mailing lists. Furuse[19] mentioned "the Internet starts with mailing lists and ends with the mailing lists".

However, conventional mailing lists are not so widely used because creating and maintaining a mailing list is not an easy task. People feel troublesome to use mailing lists for casual uses such as "arranging a holiday travel" and "talking about today's party". In this chapter, we propose a simple and powerful mailing list system called *QuickML*, with which people can easily create a mailing list and control the member account only by sending email messages. Using QuickML, people can enjoy group communication at any place, at any time, and by anyone.

5.2 Problems of Conventional Mailing Lists

Chapan[14] mentioned that he created *Majorodomo*, one of the most long-used mailing list systems, to ease the maintenance of mailing lists. Before creating Majordomo, he

was often disturbed by the following requests from participants and applicants of his mailing lists.

- I want to join your mailing list.
- I want to leave your mailing list.
- Please tell me which of your lists am I on?

The foremost purpose of Majordomo was to handle these routine requests automatically. Majordomo introduced the concept of *command mail* for managing the member account of mailing lists. The command mail is a mail that Majordomo can receive and interpret. For example, if you want to join a mailing list foo@example.com operated by Majordomo, you can send a mail including "subscribe foo" in its body to Majordomo@example.com. In this way, Majorodomo automatically handles managements of mailing lists such as joining and leaving of members. Similar mailing list systems such as *fml*[18] and *Mailman*[13] also provide command-mail-based methods.

However, these systems are not so easy to use from the viewpoint of a system administrator, since the systems require a privilege to install a new software to a mail server and also require a high-grade know-how of system administration of a mail server. Moreover, it is rather troublesome to run a special command or edit a configuration file to just create a new mailing list.

In addition, these mailing list systems are also not so easy from the viewpoint of a casual user either, since command mails are difficult for users to remember, and people often send wrong command mails which are accidentally distributed to all the members of a mailing list. In addition, it is also troublesome for users that the syntax of a command mail is different for each mailing list system.

In recent years, web-based mailing list services such as *Yahoo Groups*[1] and *FreeML*[2] become popular because these services made it easier to create and manage a mailing list for casual users. Using the services, anybody can easily create a new mailing list without a privilege of system administration. However, since these systems require a web browser as a front-end, it is difficult to use the address book of user's favorite mail client smoothly and filling in web-based forms is thus not easy. It

[1]http://groups.yahoo.com/
[2]http://freeml.com/

is also not an easy task to fill in the web-based forms especially using mobile devices such as mobile phones.

As discussed above, so far there are many troubles for using mailing lists and people often consider that mailing lists are useful only for a long term purpose such as a discussion in a large project. In other words, people feel troublesome to use mailing lists for casual uses such as “arranging a holiday travel” and “talking about today’s party”.

5.3 QuickML

We propose a new mailing list system called *QuickML*[38, 57], which solves the problem described in the previous section. QuickML reduces the cost of creation and management of mailing lists so that people can enjoy group communication very easily at any place, and at any time.

5.3.1 Usage of QuickML

QuickML allows a user to easily create and enjoy a mailing list just send an email message to a QuickML server without configuring a mail server or launching a web browser. In this section, we show how a new mailing list is created, and how member account can be easily managed using QuickML.

Creating a New Mailing List

A user can create a new mailing list simply by sending an email message to the mailing list address he or she wants to create. In the example shown below, Alice (alice@example.com) can create a new mailing list “party@quickml.com” for a party announcement, making Bob (bob@example.com) as a member of the mailing list. Just by sending this email message, a new mailing list account “party@quickml.com” is created automatically.

```
Subject: Party tonight!
To: party@quickml.com          ⇐ Address of the new ML
From: alice@example.com        ⇐ Creator of the ML
Cc: bob@example.com            ⇐ Other new members

I just created a mailing list
for our party tonight.
- Alice
```

If someone else is using “party@quickml.com” already, an error message is sent back to Alice, saying that the address is not available. In that case, Alice can use a name with a subdomain name such as “party@alice.quickml.com” to avoid the conflict.

Sending a Message to a Mailing List

If a member sends a message to the newly created mailing list address, the message is sent to all the members. The following message is delivered to both Alice and Bob.

```
Subject: [party:1] Re: Party tonight!
To: party@quickml.com          ⇐ Address of the ML
From: bob@example.com          ⇐ Member's address

What shall we bring today?
- Bob
```

Adding a New Member

If a new user is listed in the Cc: field, he or she will be added to the mailing list.

```
Subject: Adding Chris
To: party@quickml.com          ⇐ Address of the ML
From: alice@example.com        ⇐ Member's address
Cc: chris@example.com          ⇐ Address of a new member

Chris will join us.
- Alice
```

We design this method so that a new member can be invited easily just by sending a usual invitation message.

When a new member is added, QuickML adds the following header to the body of a mail to inform that a new member is joined.

```
ML: party@quickml.com
New Member: chris@e...
```

Additionally, QuickML also adds information of the list of members to the end of a mail. QuickML obfuscates the addresses of members to prevent a drain of the addresses for abuses.

```
Members of <party@quickml.com>:
alice@e...
bob@p...
chris@e...
```

Joining a Mailing List

A new user can join an existing mailing list by specifying one of the members of the mailing list in the Cc: field of the message.

```
Subject: Let me join
To: party@quickml.com                ⇐ Address of the ML
From: eve@example.com                ⇐ A person who wants to join
Cc: bob@example.com                  ⇐ Member's address

I want to join you!
- Eve
```

Since the mail sent for joining a mailing list is distributed to all members of the mailing list, a concise self-introduction is preferable to be included. We design this method so that one can join easily just by sending a usual greeting message.

Leaving a Mailing List

A member can leave the mailing list by sending an empty message to the list.

```
Subject: Bye!
To: party@quickml.com                ⇐ Address of the ML
From: bob@example.com                ⇐ Member's address
                                     ⇐ (Empty message)
```

Although sending an empty mail could be considered as a kind of command mail, it is much easier than remembering commands such as "unsubscribe enkai@quickml.com" (Majordomo) and "# bye" (fml). QuickML exceptionally accepts a command mail including just "bye" in the first line for supporting web-based mail services that adds an advertisement to the end of a mail (i.e. a user of such services cannot send an empty mail).

It is worth noting that QuickML do not notify other members when a member leaves. We chose the policy because some people often want to leave a mailing list in silence without bothering other members.

Returning to a Mailing List

A former member can rejoin the mailing list after leaving the mailing list, just by sending a message to the mailing list again.

```
Subject: I'm back!
To: party@quickml.com                ⇐ Address of the ML
From: bob@example.com                ⇐ Member's address

I'm back again!
- Bob
```

Removing a Member

To remove a member from a mailing list, any member can send an empty message to the mailing list with the member to be deleted in the Cc: field.

Subject: Remove	
To: party@quickml.com	⇐ Address of the ML
From: alice@example.com	⇐ Member's address
Cc: spam@spam.com	⇐ Address to remove
	⇐ (Empty message)

This method is especially useful for removing a member added accidentally. Since a notification mail is sent to the removed member, it is unlikely that a malicious member abuses this method for removing other member.

Comparison to Other Systems

As discussed above, since a user can perform every tasks just by sending an email, it's quite easy to create a mailing list and enjoy group communication with QuickML. Table 5.1 compares basic functions of various mailing list systems. Since we designed QuickML with special emphasis on simplicity and ease-of-use, no customization functions such as a privilege management are intentionally omitted.

5.3.2 Effective Uses

Since creating a mailing list with QuickML is very easy, people can instantly create mailing lists for casual purposes, which are never created with conventional mailing list systems. Here, we show some effective uses of QuickML.

Communication in Small Groups

Using QuickML for communication in small groups such as friends, a family, a club, colleagues, a seminar, and participants of a party is particularly useful. We have used QuickML for such purposes including a meeting and a research project.

Discussion on Minor Topics

People often hesitate to continue a discussion on a minor topic that appeals only a small number of members in a large mailing list. Using QuickML, it's very easy to create a small mailing list for continuing the discussion with the small number of members who are interested in the topic without disturbing other members.

	Manual Management	ML Management System (e.g. Majordomo)	Web-based Management (e.g. Yahoo Groups)	**QuickML**
Creating a ML[1] [1]"ML" stands for Mailing List.	× Request an administrator to create it.	× Login to a mail server and run a special command.	○ Access a web site and fill its forms.	◎ Send a mail to *new-ml-name*@QuickML.com.
Deleting a ML	× Request an administrator to delete it.	× Login to a mail server and run a special command.	○ Access a web site and fill its forms.	◎ Disappear automatically when no message is posted for more than one month.
Member Management (e.g. Adding or removing a member)	× Request an administrator to add or remove a member.	△ Send a mail including a special command to a specific address.	○ Access a web site and fill its forms.	◎ Send a mail including an address to add or remove in Cc: field.
Joining/Leaving a ML	× Request an administrator to edit the list of members.	△ Send a mail including a special command to a specific address.	○ Access a web site and fill its forms.	◎ Send a mail including an address of an existing member in Cc: field to join. Send an empty mail to leave.
Customization (Privilege control, archiving, checking a list of members, etc.)	× Request an administrator to do all of them.	○ Use a command mail to do customizations.	◎ Use forms in a web site and do customizations flexibly.	× No such customizations functions are available.
Summary	Everything needs an administrator who kindly helps. Not convenient at all.	Mastering special commands requires a technical skill. However, this system is suited for managing a large-scale mailing list with careful customization.	Managing a mailing list using a web browser is convenient. However, the web-based system lacks interoperability with users's address books in their favorite mail clients.	QuickML is convenient because everything can be done by just sending a mail. However, QuickML is not suited for operating a large-scale mailing list because it lacks customization functions.

Table 5.1: Comparison of basic functions with various mailing list systems.

5.3.3 Disadvantages

While QuickML has advantages for easily creating and using mailing lists, it also has some disadvantages.

Large Scale Mailing List

QuickML is especially suitable for small scale mailing lists. However, it is unsuitable for large scale mailing lists, since QuickML lacks customization functions that satisfies a large number of members. For such purposes, conventional mailing list systems such as Majordomo are more suitable.

Security

On the one hand, conventional mailing list systems such as Majordomo have a function for confirming a new member whether he or she really wants to join or not. This function prevents a malicious person from adding a third-party address to a mailing list without permission. While the function increases security level, it also decreases easiness of joining to a mailing list.

On the other hand, we omitted the function of confirmation from QuickML for the sake of easiness. While worries about security remain, troubles are hardly happened for two years of our public service.

5.4 Implementation

A QuickML server runs as a SMTP (Simple Mail Transfer Protocol)[28] server, and it accepts most of requests defined in SMTP. To make the QuickML server simple, it does not perform various MTA jobs including delivering messages and handling errors, but it delegates most of these tasks to existing MTAs such as *sendmail* and *Postfix*. When a QuickML server receives an email message, it first checks if the message requests administrative tasks like creating a new mailing list, and performs the task according to the request. Then it delegates all the mail transfer tasks to an existing MTA.

5.4.1 Automatic Administration of Mailing Lists

Automatic Removal of Members

Conventional mailing list systems such as Majordomo report all the errors to the administrator, and the administrator has to receive many error mails every day. Since the format of the error mails is not standardized[3], the administrator has to read the error mails and take appropriate actions. For example, he or she has to distinguish a temporarily unreachable address from a permanently unreachable address such as a wrong address. To eliminate these chores, the QuickML server processes error mails to remove unreachable mail addresses from mailing lists automatically.

The QuickML server automatically removes a mail address from a mailing list when it receives error mails five times from the same address. It is notable that if the QuickML server counts a number of error mails naively, a temporarily unreachable address could be removed accidentally in a mailing list with much traffic. For this reason, we designed the QuickML server not to count error mails received within 24 hours from the last error mail. Thus, users have not to worry about the automatic removal if their mail server is temporarily unavailable. In addition, the counter of error mails is reset to zero when a mail is posted from the corresponding mail address.

We implemented the automatic error mail handling mechanism by using *VERP*[10] extension of qmail[11] and Postfix[58] mail servers. Since the VERP extension adds an unique ID to "envelope from" [4] of each recipient address, the QuickML server can easily identify error mails.

Automatic Disappearance of Mailing Lists

A QuickML-based mailing list disappears when all members leave. Since there is no administrator in a QuickML-based mailing list, even a member who creates a mailing list can leave and the mailing list continues without the original member.

If no message is posted for one month, the mailing list will also disappear. A notification mail is sent to all members a week before the day of disappearance. If no message is still posted for a week, the mailing list will disappear.

[3] An extension to SMTP[41] is proposed but not popularized yet.

[4] A sender's address appears only in a "MAIL FROM:" line of SMTP.

This automatic disappearance solves a problem of conventional mailing list systems, that inactive mailing lists last forever even if no message is posted for years. In this way, users can easily create a mailing list using QuickML without caring about the lifetime of their mailing list.

Handling of Looping Mails

A mailing list system should prevent looping of a mail. Looping of a mail happens when a mail posted to a mailing list is accidentally returned. QuickML prevents the loop by the following rules.

- When QuickML delivers a mail to the members of a mailing list, QuickML adds a special mark to the header of the mail and QuickML rejects a mail containing the mark.
- QuickML rejects an address of a mailing list created by QuickML itself as a member of a mailing list.

Handling of Subdomains

One of the most effective features of QuickML server is automatic subdomain handling. Anyone can create a mailing list with arbitrary subdomain name. QuickML server employs DNS's wildcard MX mechanism[40] to gather all incoming mails to any subdomains into a host where QuickML server is running. The following snippet illustrates a sample configuration of a BIND name server to run QuickML server at qml.example.com (192.168.0.1) accepting all incoming mails to any subdomains of example.com.

```
$ORIGIN example.com.
@      IN MX 10 qml
*      IN MX 10 qml
qml    IN A     192.168.0.1
       IN MX 10 qml
```

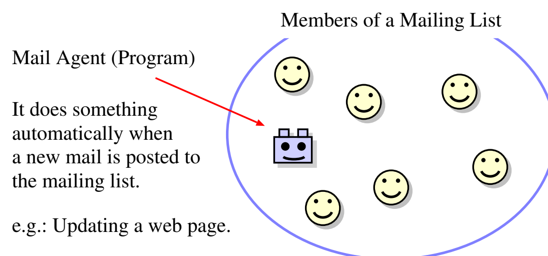


Figure 5.1: Illustration of the concept of mail agents.

5.4.2 Using Agents with QuickML

Since we implement QuickML as simple as possible, QuickML does not have functions usually provided by conventional mailing list systems, such as mail archiving. Although adding such functions to QuickML is not so hard, we deliberately avoid to add for simplicity.

Instead, we choose to keep QuickML simple, and introduce a *mail agent* that works separately from the QuickML server. A mail agent is a program that has a mail address and a user can use the mail agent by adding it as a member of a mailing list. Figure 5.1 illustrates the concept of the mail agent.

To implement mail archiving, we create a mail agent called *archiver agent*, instead of adding the equivalent function to the QuickML server. The archiver agent stores mails posted to a mailing list and creates web pages so that one can read the mail archive using a web browser.

Figure 5.2 shows a mail archive created by the archiver agent. The archiver agent has sufficient functions for mail archiving. If a user wants a new function such as message threading in web pages, he or she can modify just this archiver agent without touching the QuickML server.

It is notable that the archiver agent is a simple example of the integration of search and communication techniques since the archiver agent provides a searchable mail archive, that is a user can search the mail archive while reading messages.

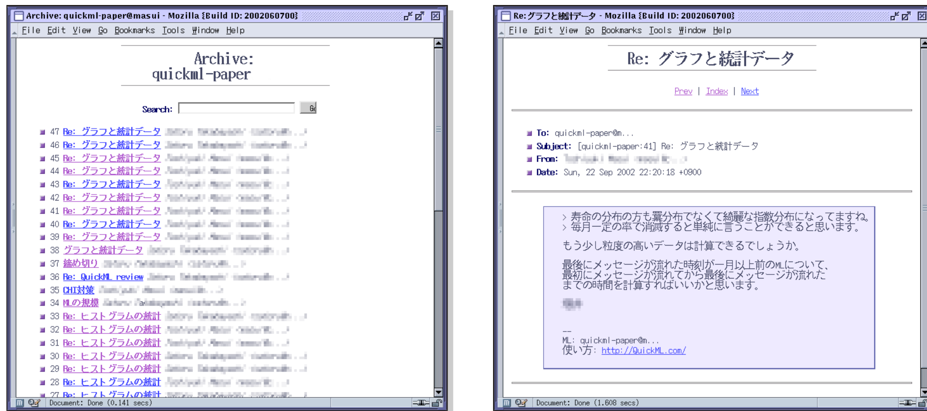


Figure 5.2: Web pages created by the archiver agent.

5.5 Discussion

We started QuickML.com in November of 2001. After testing the service with a small group for two months, we launched the public service of QuickML and over 55,000 people are using QuickML.com at the end of May 2003. In this section, we discuss experiences gained by the public service and statistical observations.

5.5.1 Experiences

Through the operation of QuickML.com, we observed several interesting usage patterns of QuickML quite different from conventional mailing list systems. Here, we discuss our experiences and the usage patterns.

Small is Beautiful

When QuickML.com started, many people requested us to provide additional functions for customization. However, several months later, we received much less requests for such functions. We interpret that people understand such functions are hardly needed for casual group communications. Omitting as many functions as possible is the key concept QuickML.

Power Users

As soon as QuickML.com started, power users appeared who created a large number of mailing lists. They created mailing lists topic by topic. This reflects a characteristic of QuickML, that creation of a mailing list is very easy. In fact, one of the users created 315 mailing lists in 18 months, and he received 31% of the messages delivered from QuickML.com every day.

Mobile Phone Users

Number of mobile phone users reaches about 25,000. In other words, they occupied about 45% of all. This fact shows that QuickML is quite suitable for group communication in mobile environments. We have been using QuickML with mobile phones especially for exchanging information while traveling and attending a conference.

Names of Mailing Lists

Before starting QuickML.com, we expected that people prefer very short names such as “a@quickml.com” and these names frequently causes collisions. However, we have not seen such very short names and collisions so much.

We also expected that people did not prefer using subdomain names in the addresses of their mailing lists. However, people preferred using subdomain names and subdomain names were used in about 25% of mailing lists possibly to avoid collisions of names of mailing lists.

Security Issues

Since a non-member cannot send a message to a mailing list of QuickML, it is difficult to send a “spam” message to existing mailing lists indiscriminately.

Although it could be possible to abuse QuickML to send a spam message by creating a mailing list for the spam purpose, we have not seen such abuses probably QuickML.com limits the number of members in a mailing list to one hundred and abusing QuickML.com for sending only one hundred spam messages does not pay off.

5.5.2 Statistical Observations

We observed various characteristics of QuickML from the statistical facts by analyzing the log data of QuickML.com.

Growth of QuickML Users

Figure 5.3 shows the number of QuickML users since November 2001. After 18 months, the number of users reached about 55,000. We can see that the number of users is growing almost linearly.

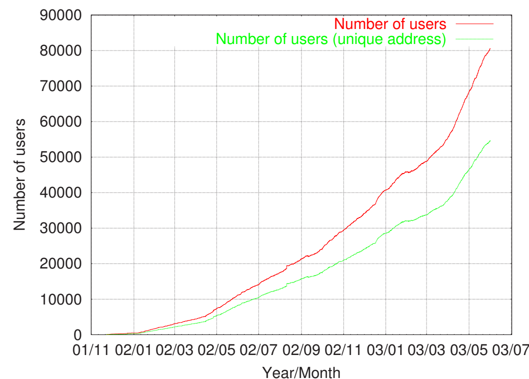


Figure 5.3: Growth of QuickML users.

Number of Posted Messages

Figure 5.4 shows the total number of messages posted to QuickML.com. The upper graph shows the total number of messages since November 2001, and we can see that 800,000 messages were posted for 18 months and average 1,200 messages were posted a day. The lower graph shows the total number of messages between May 25th and May 31st and we can observe that the graph reflects the rhythm of users's life from the curve that they did not send messages so much in the middle of the night.

Number of Mailing Lists

Figure 5.5 shows the number of mailing lists. For 18 months, over 14,000 mailing lists were created, and nearly half of them were closed. This reflects particularly the

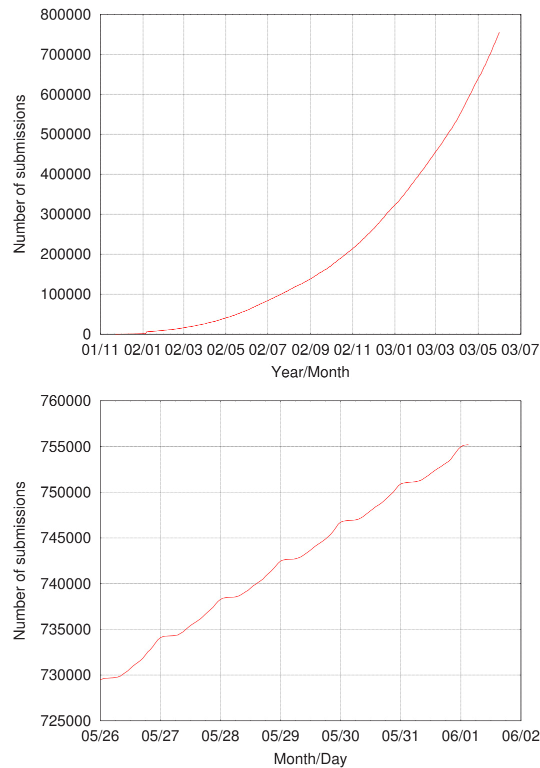


Figure 5.4: Number of posted messages.

characteristic of QuickML that mailing lists automatically close when no message is posted for one month.

Lifetime of Mailing Lists

Figure 5.6 shows the distribution of the lifetime of mailing lists. From the upper graph, we can clearly see that most of mailing lists disappear in one or two months. The lower graph shows the log plot of the same data. The graph curves almost linear and we can see that the graph obeys the exponential distribution.

Number of Members of Mailing Lists

Figure 5.7 shows the distribution of the numbers of members in the mailing lists. We can see that mailing lists with four to seven members are most common, and the distri-

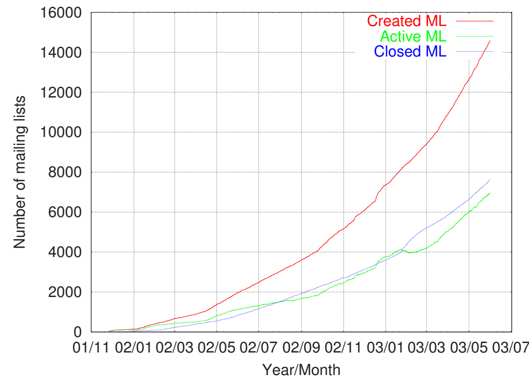


Figure 5.5: Number of mailing lists.

bution almost obeys the logarithmic normal distribution function shown by the dotted line. Note that we omitted the data of mailing lists with only one member to draw the dotted line because many of these mailing lists seems to be created for test or by mistake.

Correlation between Size and Lifetime of Mailing Lists

Figure 5.8 shows the correlation between the size of mailing lists and their lifetime. The "lifetime" of a mailing list means the time period between the first message and the last message posted to the mailing list, and it can be very short if the mailing list is used only temporarily. We can see that mailing lists with many members tend to live longer than small mailing lists.

5.6 Related Work

Wiki Wiki Web[32] (Wiki for short) is a web-based system that allows anyone to create and edit web pages using a web browser. Recently, Wiki is becoming popular for cooperative writing as well as casual group communication on the web. On the one hand, wiki is quite different from QuickML because Wiki is a web-based system. On the other hand, Wiki is quite similar to QuickML in that anyone can easily create a forum for group communication without an administrator who has privileges.

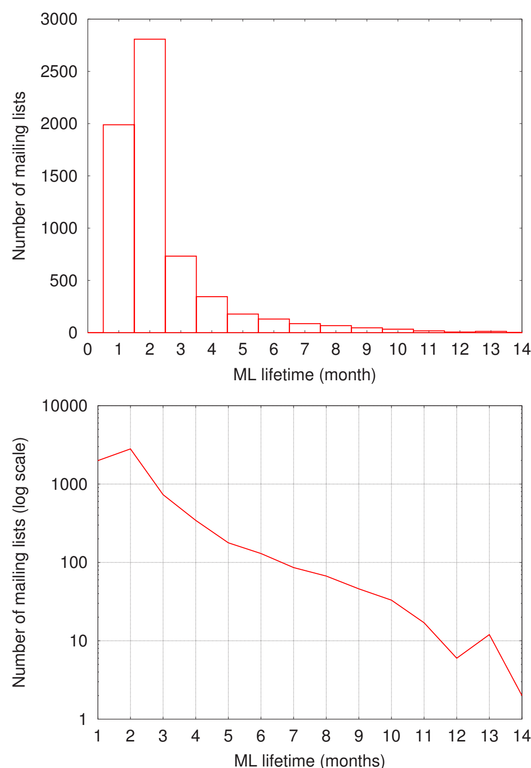


Figure 5.6: Distribution of lifetime of mailing lists.

Dey[15] proposed a toolkit for constructing context-aware applications, and created a location-based mailing list system using the toolkit, where email messages are sent to the members who are currently in a building. Location-based casual communication would be very important in the ubiquitous computing environment, and we are palling to use location information for QuickML.

Sakata[49] pointed out that conventional mailing list systems are sometimes cumbersome because uninteresting messages are sent to all members whether one likes it or not. To solve the problem, Sakata proposes a session-based mailing list system that can quickly create a new mailing list for interested members only. Using the system, a user can call for participants for a new topic in existing mailing lists and create a sub mailing list of the topic with the members who answer the call. This system is similar to QuickML in the aim for making casual group communication easy but they are quite different in the way of creating a mailing list. For example, a user can create

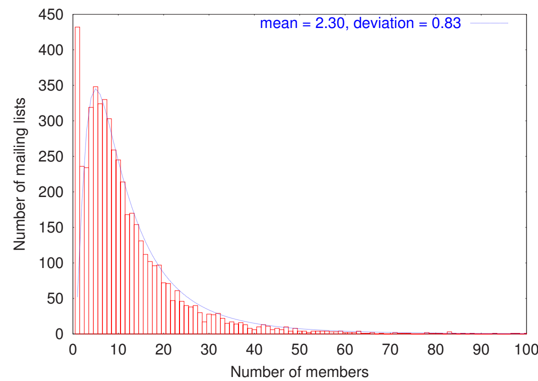


Figure 5.7: Distribution of number of members.

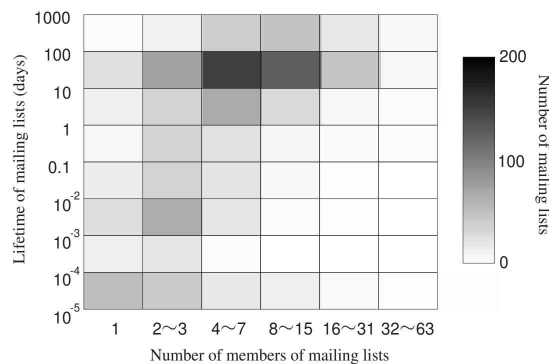


Figure 5.8: Correlation between size and lifetime of mailing lists.

a new mailing list by just sending an email with QuickML while a user have to access a web-based system to create a new mailing list with the system proposed by Sakata.

5.7 Summary

We propose a simple and powerful mailing list system called QuickML which allows a user to easily create a mailing lists only by sending an email. Using QuickML, people can enjoy group communication at any place, at any time.

Although real-time communication media such as an instant messenger have become popular recently, we think that email will not go out of fashion as a asynchronous

communication medium and mailing lists remain useful as long as email are used. Through the experience of using QuickML for over a year, we cannot think doing group communication on the Internet without QuickML.

6

Conclusion

The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore, all progress depends on the unreasonable.

— George Bernard Shaw

This thesis has presented various methods and applications for creation and sharing of information.

In Chapter 2, we briefly described background search techniques and their characteristics of them. We emphasize that a suffix array is especially useful for applied text processing using large corpora. We also propose search systems called Namazu and Sary to investigate these search techniques and make good use of them for practical uses. It is notable that our search systems are both open to the public as free softwares and widely used by a large number of users.

In Chapter 3, we realize incremental search for Japanese text, which was hard to perform effectively before. We also show how Migemo effectively improves the search operations through evaluation conducted with subjects who use Migemo regularly. The result show that users editing Japanese documents do incremental searches in Japanese nearly one third overall and Migemo helps the users effectively. Migemo is also open to the public as a free software and Migemo has been ported to several systems including a text-based web browser w3m, a text editor VIM, and a text editor xyzzy by other developers. We think that these ports of Migemo are evidences of the usefulness of our method.

In Chapter 4, we present several methods for writing assistance as well as our back-

bone philosophy “Writing is Searching”. We regard writing as a process of searching for words or expressions, and consider that writing is, in a broad sense, a process of searching. While we propose several methods for writing assistance including input acceleration systems and proofreading systems, we also have been actively interacted with other researchers. One effort made by a collaboration is a new predictive text input system using a large number of self-written documents[30]. We hope that we can continue the research with a broader range of researchers.

In Chapter 5, we propose a group communication system called *QuickML* that allows users to share information with others easily, and show how QuickML effectively help users enjoy casual group communication through the public service involving thousands of users. With QuickML, people can easily create a mailing list and control the member account only by sending email messages. Since creating a mailing list with QuickML is very easy, people can instantly create mailing lists for casual purposes, which are never created with conventional mailing list systems. We started the public service of QuickML in 2001 and over 55,000 people were using QuickML.com by the end of May 2003 and mobile phone users occupied 45% of all. This fact shows that QuickML is quite suitable for group communication in mobile environments.

While the author believes that these methods and applications help people create and share information substantially, there are many scopes of improvements.

First, media types other than a text should be supported. In fact, our recent research interest is not limited to textual communication but we also has investigated communication methods for other types of media such as image and music. Second, more cooperation of search and communication is needed. While we introduced the archiver agent described in Chapter 5, we have been working on other applications. One example of such challenges is a chat assistance system. The system learns a user’s chat log and assists a conversation in several ways including predicting appropriate short responses like “I see” and “uh-huh” so that a user can simply click one of them to give a response without touching a keyboard. We are planning to extend the system for further assistance such as presenting information related to the conversation in real-time by employing search techniques using both personal information and resources on the Internet.

For more effective creation and sharing of information, further research should be carried out. This work is only the first step to our future research project.

References

[1] SKK openlab. http://openlab.jp/skk/.

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

[3] Philip E. Agre. My top 10 email hassles. *Communications of the ACM*, Vol. 38, No. 7, p. 122, 1995.

[4] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, Vol. 18, No. 6, pp. 333–340, 1975.

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

[6] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, Vol. 35, No. 10, pp. 74–82, 1992.

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

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

[9] 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.

[10] Dan Bernstein. Variable envelope return paths, 1997. http://www.jp.qmail.org/qmaildoc/RFC/RFCVERP.html.

[11] Dan Bernstein. qmail: #3 mta on the internet, 2002. http://www.qmail.org/.

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

[13] Mailman Cabal. Mailman, the GNU mailing list manager. http://www.list.org/.

[14] D. Brent Chapan. Majordomo: How I manage 17 mailing lists without answering "–request" mail. In *LISA VI The Systems Administration Conference*, 1992.

[15] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Vol. 16, pp. 97–166, 2001.

[16] Nicolas Ducheneaut and Victoria Bellotti. E-mail as habitat: an exploration of embedded personal information management. *interactions*, Vol. 8, No. 5, pp. 30–38, 2001.

[17] Jeffrey E. F. Friedl. *Mastering Regular Expressions, 2nd Edition*. O'Reilly, 2002.

[18] Ken'ichi Fukamachi. *fml bible*. O'Reilly Japan, 2001. (in Japanese).

[19] Yukihiro Furuse and Katsuya Hirose. *The World Changed through the Internet*. Iwanami Shoten, 1996. (in Japanese).

[20] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Processings of the ACM Symposium on Theory of Computing (STOC2000)*, pp. 397–406, 2000.

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

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

[23] Peter E. Hart and Jamey Graham. Query-free information retrieval. In *Proceedings of the Conference on Cooperative Information Systems*, pp. 36–46, 1994.

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

[25] Hideo Itoh. A method for segmenting japanese text into words by using suffix array. In *Information Processing Society of Japan, Special Interest Group on Natural Language Processing (IPSJ-SIGNL) 99-NL-131*, pp. 47–54, 1999. (in Japanese).

[26] Hideo Itoh. *Studies on String Indexing and Its Application to Natural Language Processing*. PhD thesis, Tokyo Institute of Technology, 2000. (in Japanese).

[27] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *Proceedings of the sixth Symposium on String Processing and Information Retrieval (SPIRE '99)*, pp. 81–88. IEEE Computer Society Press, 1999.

[28] Editor J. Klensin. RFC2821: Simple mail transfer protocol, 2001. ftp://ftp.isi.edu/in-notes/rfc2821.txt.

[29] 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.

[30] Hiroyuki Komatsu, Satoru Takabayashi, and Toshiyuki Masui. Predictive text input using document storage system "kukura". In *Proceedings of Workshop on Interactive Systems and Software (WISS2002)*, pp. 43–47, 2002. (in Japanese).

[31] Hiroyuki Komatsu, Satoru Takabayashi, and Toshiyuki Masui. Context-aware predictive text input method using dynamic abbreviation expansion. *Information Processing Society of Japan (IPSJ) Journal*, Vol. 44, No. 11, pp. 2538–2546, 2003. (in Japanese).

[32] Bo Leuf and Ward Cunningham. *Wiki Way, The: Quick Collaboration on the Web*. Addison-Wesley, 2001.

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

[34] 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.

[35] Toshiyuki Masui. An efficient text input method for pen-based computers. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI '98)*, pp. 328–335. Addison Wesley, 1998.

[36] 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.

[37] Toshiyuki Masui. Q-Pocket: A new approach to personal information management. In *Interactive Systems and Software VIII: Japan Society for Software Science and Technology WISS2000*, pp. 191–196, 2000. (in Japanese).

[38] Toshiyuki Masui and Satoru Takabayashi. Instant group communication with QuickML. In *Proceedings of the ACM Conference on Supporting Group Work (Group '03)*, pp. 268–273, 2003. (in Japanese).

[39] Yuji Matsumoto. Morphological analysis system chasen. *Information Processing Society of Japan (IPSJ) Magazine*, Vol. 41, No. 11, pp. 1208–1214, 2000. (in Japanese).

[40] Paul Mockapetris. RFC1034: Domain names - concepts and facilities, 1987. ftp://ftp.isi.edu/in-notes/rfc1034.txt.

[41] Keith Moore. RFC1891: SMTP service extension for delivery status notifications, 1996. ftp://ftp.isi.edu/in-notes/rfc1891.txt.

[42] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, Vol. 33, No. 1, pp. 31–88, 2001.

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

[44] T-Code Open Laboratory Project. T-Code Laboratory. (in Japanese).

[45] Jef Raskin. *The Humane Interface*. Addison Wesley, 2000.

[46] 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.

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

[48] Kunihiko Sadakane. Indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, Vol. 48, No. 2, pp. 294–313, 2003.

[49] Kazuhiro Sakata and Akihisa Kurashima. A mailing list system enabling consummatory communication. *Information Processing Society of Japan (IPSJ) Journal*, Vol. 41, No. 10, pp. 2762–2769, 2000. (in Japanese).

[50] Satoshi Sato. Ctm: An example-based translation aid system. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING '92)*, pp. 1259–1263, 1992. (in Japanese).

[51] Ben Shneiderman. Dynamic queries for visual information seeking. *IEEE Software*, Vol. 11, No. 6, pp. 70–77, 1994.

[52] Richard Stallman. *GNU Emacs Manual*. Free Software Foundation, 2000.

[53] Satoru Takabayashi. Namazu: Yet another full-text search engine for vast quantities of documents. *Information Processing Society of Japan (IPSJ) Magazine*, Vol. 41, No. 11, pp. 1227–1232, 2000. (in Japanese).

[54] Satoru Takabayashi. Sary: Reusable components and tools for searching large corpora. In *Demonstration at the Second Meeting of the North American Chapter of the Association for Computational Linguistics*, pp. 100–101, 2001.

[55] Satoru Takabayashi, Hiroyuki Komatsu, and Toshiyuki Masui. Migemo: Incremental search method for languages with many character faces. In *Proceedings of the 6th Natural Language Processing Pacific Rim Symposium*, pp. 435–438, 2001.

[56] Satoru Takabayashi, Hiroyuki Komatsu, and Toshiyuki Masui. Migemo: Incremental method for japanese text. *Information Processing Society of Japan (IPSJ) Journal*, Vol. 43, No. 12, pp. 3698–3705, 2002.

[57] Satoru Takabayashi and Toshiyuki Masui. Instant group communication with QuickML. *Information Processing Society of Japan (IPSJ) Journal*, Vol. 44, No. 11, pp. 2608–2616, 2003.

[58] Wietse Venema. The Postfix home page, 2002. http://www.postfix.org/.

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

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

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

[62] Tatsuo Yamashita. SUFARY home page. (in Japanese).

[63] Tatsuo Yamashita. Morphological analysis directly using pos-tagged corpora. In *4th Annual meeting of Association of Natural Language Processing*, pp. 524–527, 1998. (in Japanese).

[64] Tatsuo Yamashita and Yuji Matsumoto. Full text approximate string search using suffix arrays. In *Information Processing Society of Japan, Special Interest Group on Natural Language Processing (IPSJ-SIGNL) 97-NL-121*, pp. 83–90, 1997. (in Japanese).

[65] Robert H Zakon. Hobbes' internet timeline - the definitive arpanet and internet history, 2003. http://www.zakon.org/robert/internet/timeline/.

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 members of the thesis committee Professor Shunsuke Uemura an Professor Minoru Itoh for carefully reading this thesis and giving me valuable comments. I especially like to thank Associate Professor Kentaro Inui. His rigorous attitude toward research as well as *Japanese Sake* influenced me highly.

My first year at NAIST was entirely satisfactory because of the firm friendship with Dr. 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 methodologies. 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 quite enlightened me.

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

I am also grateful to Dr. Toshiyuki Masui for giving me precious suggestions. His works inspire this work very much.

Special thanks are reserved for Hiroyuki Komatsu and Koji Tsukada. They always support me in various ways both technical and personal when I'm in trouble.

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

List of Publication

Journal Papers

1. Satoru Takabayashi, Toshiyuki Masui. Instant Group Communication with QuickML. (in Japanese) Information Processing Society of Japan (IPSJ) Journal, Vol. 44, No. 11, pp. 2608-2616, 2003.

2. Satoru Takabayashi, Hiroyuki Komatsu, Toshiyuki Masui. Migemo: Incremental Search Method for Japanese Text. (in Japanese) Information Processing Society of Japan (IPSJ) Journal, Vol. 43, No. 12, pp. 3698–3705, 2002.

3. Satoru Takabayashi. Namazu: Yet Another Full-Text Search Engine for Vast Quantities of Documents. (in Japanese) Information Processing Society of Japan (IPSJ) Magazine, Vol. 41, No. 11, pp. 1227–1232, 2000.

4. Hiroyuki Komatsu, Satoru Takabayashi, Toshiyuki Masui. Context-aware Predictive Text Input Method Using Dynamic Abbreviation Expansion. (in Japanese) Information Processing Society of Japan (IPSJ) Journal, Vol. 44, No. 11, pp. 2538-2546, 2003.

International Conferences

1. Satoru Takabayashi, Hiroyuki Komatsu, and Toshiyuki Masui. Migemo: Incremental search method for languages with many character faces. In *Proceedings of the 6th Natural Language Processing Pacific Rim Symposium*, pp. 435–438, 2001.

2. Satoru Takabayashi. Sary: Reusable components and tools for searching large corpora. In *Demonstration at the Second Meeting of the North American Chapter of the Association for Computational Linguistics*, pp. 100–101, 2001.

3. Koji Tsukada, Satoru Takabayashi, and Toshiyuki Masui. Dying link. In *Proceedings of the 10th International Conference on Human-Computer Interaction (HCI 2003)*, Vol. 3 (Human-Central Computing), pp. 1353–1357, 2003.

4. Toshiyuki Masui, Satoru Takabayashi. Instant Group Communication with QuickML In *Proceedings of the ACM Conference on Supporting Group Work (Group '03)*, pp. 268–273, 2003.

Other Publications

1. Satoru Takabayashi, Koji Tsukada, Toshiyuki Masui. FaceIcon: A Simple File Transfer System. (in Japanese) In *Proceedings of Interaction 2003*, pp. 33–34, 2003.

2. Satoru Takabayashi, Toshiyuki Masui. Instant Group Communication with QuickML. (in Japanese) In *Proceedings of Workshop on Interactive Systems and Software (WISS2002)*, pp. 1–8, 2002.

3. Satoru Takabayashi, Toshiyuki Masui. QuickML: A Simple Mailing List System. (in Japanese) Information Processing Society of Japan, Special Interest Group on Human Interface (IPSJ-SIGHI) 2002-HI-100, pp. 71–78, 2002.

4. Satoru Takabayashi, Hiroyuki Komatsu, Toshiyuki Masui. Migemo: Incremental Search Method for Japanese Text. (in Japanese) Information Processing Society of Japan, Special Interest Group on Human Interface (IPSJ-SIGHI) 2001-HI-94, pp. 41–46, 2001.

5. Satoru Takabayashi, Yuji Matsumoto. Writing Assistance through Search Techniques. (in Japanese) In *Proceedings of the 7th Annual meeting of Association of Natural Language Processing*, pp. 127–130, 2001.

6. Satoru Takabayashi. Writing Assistance through Search Techniques. Master's thesis, Nara Institute of Science and Technology, 2001.

7. Hiroyuki Komatsu, Satoru Takabayashi, Toshiyuki Masui. Text Input using Dynamic Abbreviation. (in Japanese) Information Processing Society of Japan, Special Interest Group on Human Interface (IPSJ-SIGHI) 2001-HI-95, pp. 133–138, 2001.

8. Hiroyuki Komatsu, Satoru Takabayashi, Toshiyuki Masui. Predictive Text Input with Japanese Dynamic Abbreviation Expansion Method "Nanashiki". (in

Japanese) In *Proceedings of Workshop on Interactive Systems and Software (WISS2001)*, pp. 67–74. 2001.

9. Hiroyuki Komatsu, Satoru Takabayashi, Toshiyuki Masui. Predictive Text Input using Document Storage System "Kukura". (in Japanese) In *Proceedings of Workshop on Interactive Systems and Software (WISS2002)*, pp. 43–47, 2002.

10. Koji Tsukada, Satoru Takabayashi. Dying Link. (in Japanese) Information Processing Society of Japan (IPSJ) Journal, Vol. 43, No. 12, pp. 3718–3721, 2002.

11. Koji Tsukada, Satoru Takabayashi. Dying Link. (in Japanese) In *Proceedings of Workshop on Interaction 2002*, pp. 73–74, 2002.

12. Toshiyuki Masui, Koji Tsukada, and Satoru Takabayashi. Information Navigation by Neighbor Hopping. (in Japanese) In *Proceedings of Workshop on Interactive Systems and Software (WISS2003)*, pp. 79–86, 2003.

Awards

1. Satoru Takabayashi. Information Processing Society of Japan (IPSJ) Yamashita SIG Research Award, 2002. Migemo: Incremental Search Method for Japanese Text.