Collaborative Report DC–97–03

Proceedings of the Prague Stringology Club Workshop '97

Edited by Jan Holub

November 1997

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Program Committee

Jun-ichi Aoe, Jan Holub, Bořivoj Melichar, Bruce W. Watson

Organizing Committee

Martin Bloch, Jan Holub

Sponsors

We thank Exprit, Fulsoft and Hewlett-Packard for their support of the Prague Stringology Club Workshop '97.

Table of contents

Preface

This collaborative report contains the proceedings of the Prague Stringology Club Workshop '97 (PSCW'97), held at the Department of Computer Science and Engineering of Czech Technical University in Prague on July 7, 1997. The workshop was preceded by PSCW'96 which was the first action of the Prague Stringology Club. The proceedings of PSCW'96 were published as a collaborative report DC–96–10 of Department of Computer Science and Engineering and are also available in the postscript form at Web site with URL: `http://cs.felk.cvut.cz/psc`. While the papers of PSCW'96 were invited papers, the papers of PSCW'97 were submitted as a response to a call for papers. The papers in this proceedings are ordered according to the sequence of their presentation.

The PSCW aims at strengthening the connection between stringology (computer science on strings and sequences) and finite automata theory. The automata theory has been developed and successfully used in the field of compiler construction and can be very useful in the field of stringology too. The automata theory can facilitate the understanding of existing algorithms and the developing of new algorithms.

Jan Holub, editor

An Efficient Trie Hashing Method Using a Compact Binary Trie

Masami Shishibori, Makoto Okada, Toru Sumitomo and Jun-ichi Aoe

Department of Information Science & Intelligent Systems
Faculty of Engineering
Tokushima University
2-1 Minami-Josanjima-Cho
Tokushima-Shi 770
Japan

e-mail: {bori, aoe}@is.tokushima-u.ac.jp

Abstract. In many applications, information retrieval is a very important research field. In several key strategies, the binary trie is famous as a fast access method to be able to retrieve keys in order. However, if the binary trie structure is implemented, the greater the number of the registered keys, the larger storage is required, as a result, the binary trie can not be stored into the main memory. In order to solve this problem, the method to change the binary trie into a compact bit stream have been proposed, however, searching and updating a key takes a lot of time in large key sets. This paper proposes the method to improve the time efficiency of each process by introducing a new hierarchical structure. The theoretical and experimental results show that this method provides faster access than the traditional method.

Key words: information retrieval, trie hashing, binary trie, data structures, pre-order bit stream

1 Introducion

In many natural language processing and information retrieval systems, it is necessary to be able to adopt a fast digital search, or trie search for looking at the input character by character. In digital search methods, trie method [1], [2], [3], [4] is famous as one of the fastest access methods, and trie searching is frequently used as a hash table of trie hashing [5] indices in information retrieval systems and dictionaries in natural language processing systems. Although hash and B-tree strategies are based on comparisons between keys, a trie structure can make use of their representation as a sequence of digits or alphabetic characters. A trie can search all keys made up from prefixes in an input string, in only one time scanning, since a trie advances the retrieval character by character, which makes up keys. From this reason, the trie is called the Digital Search-tree (DS-tree). Especially, DS-tree whose nodes have only two arcs labelled with 0 and 1 is called a Binary Digital Search-tree (BDS-tree) [5], [6].

In the case when the binary trie, that is BDS-tree, is implemented as the index of information retrieval application, if the key sets to be stored are large, it is too big to store into main memory. Therefore, it is very important to compress the binary trie into a compact data structure. Then, Jonge et al. [5] proposed the method to compress the binary trie into a compact bit stream, which is called the pre-order bit stream, by traversing the trie in pre-order. However, the bigger the binary trie, the longer the pre-order bit stream is, as a result, the time cost to retrieve keys located toward the end of the bit stream is high.

This paper proposes a new method able to avoid the increase of the time-cost even if the dynamic key sets become very big. The data structures compressed by this method have two distinctive features: (1) they store no pointers and require one bit per node in the worst case, and (2) they are divided into the small binary tries, and their small tries are connected by pointers.

2 A Compact Data Structure for Binary Tries

In the BDS-tree, the binary sequence, which is obtained from the translation of the characters into their binary code, is used as the value of the key, namely, the left arc is labeled with the value '0' and the right arc with the value '1'. If each of leaves in the BDS-tree points the record of only one key, the depth of the BDS-tree becomes very deep. So, each leaf has the address of the bucket, where some corresponding keys to the path are stored. We will use B_SIZE to denote the number of keys and their records that can be stored in one bucket. For example, let us suppose that the following key set K is inserted into the BDS-tree.

$$K = \{\text{air, art, bag, bus, tea, try, zoo}\}$$

If the binary sequence, obtained from the translation of the internal code of each character, where internal codes of a, b, c, z are 0, 1, c, 25 respectively, into binary numbers of 5 bits, is used, the corresponding bit strings to be registered are below.

$$\begin{array}{rl}
\text{air} \rightarrow & 0/\ \ 8/\ 17 \rightarrow 00000\ 01000\ 10001 \\
\text{art} \rightarrow & 0/\ 17/\ 19 \rightarrow 00000\ 10001\ 10011 \\
\text{bag} \rightarrow & 1/\ \ 0/\ \ 6 \rightarrow 00001\ 00000\ 00110 \\
\text{bus} \rightarrow & 1/\ 20/\ 18 \rightarrow 00001\ 10100\ 10010 \\
\text{tea} \rightarrow & 19/\ \ 4/\ \ 0 \rightarrow 10011\ 00100\ 00000 \\
\text{try} \rightarrow & 19/\ 17/\ 24 \rightarrow 10011\ 10001\ 11000 \\
\text{zoo} \rightarrow & 25/\ 14/\ 14 \rightarrow 11001\ 01110\ 01110
\end{array}$$

If B_SIZE is 2, the corresponding BDS-tree for the key set K is shown in Figure 1. In order to compress the BDS-tree, we applied the particular leaf which does not have any addresses for the bucket. This leaf will be called dummy leaf. Using the dummy leaf, the following advantages are derived. First, it satisfies the property of binary trees that the number of leaves is one more than the number of internal nodes. This property underlies the search algorithm using the compact data structure. Next, if the search terminates in a dummy leaf, the search key is regarded as a key that does not belong to the BDS-tree, and no disk access at all will be needed.

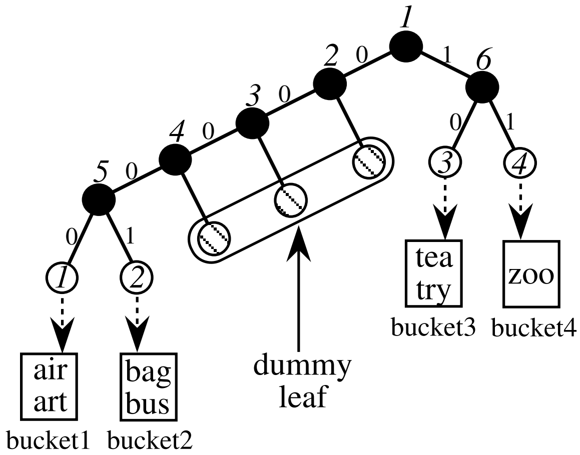


Figure 1: An Example of the BDS-tree.

When the BDS-tree is implemented, the larger the number of the registered keys, the greater the number of the nodes in the tree is, and more storage space is required. So, Jonge et al. [5] proposed the method to compress the BDS-tree into a very compact bit stream. This bit stream is called pre-order bit stream. The pre-order bit stream consists of 3 elements: *treemap*, *leafmap* and *B_TBL*. The *treemap* represents the state of the tree and can be obtained by a pre-order tree traversal, emitting a '0' for every internal node visited and a '1' for every bucket visited. The *leafmap* represents the state (dummy or not) of each leaf and by traversing in pre-order the corresponding bit is set to a '0' if the leaf is dummy, otherwise the bit is set to a '1'. The *B_TBL* stores the addresses of each bucket. Figure 2 shows the pre-order bit stream corresponding to the BDS-tree of Figure 1. Then, in order to understand the relation between the BDS-tree and the pre-order bit stream easily, we indicate above the *treemap* the corresponding internal node and leaf number (in the case of the dummy leaf, the symbol is a "d") within the round "()" and square "[]" brackets, respectively.

The search using the pre-order bit stream proceeds bit by bit from the first bit of *treemap*, so that the search is traversed the BDS-tree in pre-order. The search algorithm using the pre-order bit stream is presented below, where it uses the following variables and functions:
s_key: The bit string of the key to be searched.
keypos: A pointer to the current position in *s_key*.
treepos: A pointer to the current position in *treemap*.
leafpos: A pointer to the current position in *leafmap*.
bucketnum: The corresponding bucket number.
SKIP_COUNT(): Skips the left partial tree, and returns the number of the leaf within the partial tree.
FIND_BUCKET(): Returns the corresponding bucket number of *s_key*.
[An Algorithm to search in the BDS-tree]
Input: *s_key*;
Output: If *s_key* can be found, then the output is TRUE, otherwise FALSE;

Step(S-1): {Initialization}
$keypos \leftarrow 1$, $treepos \leftarrow 1$, $leafpos \leftarrow 1$;

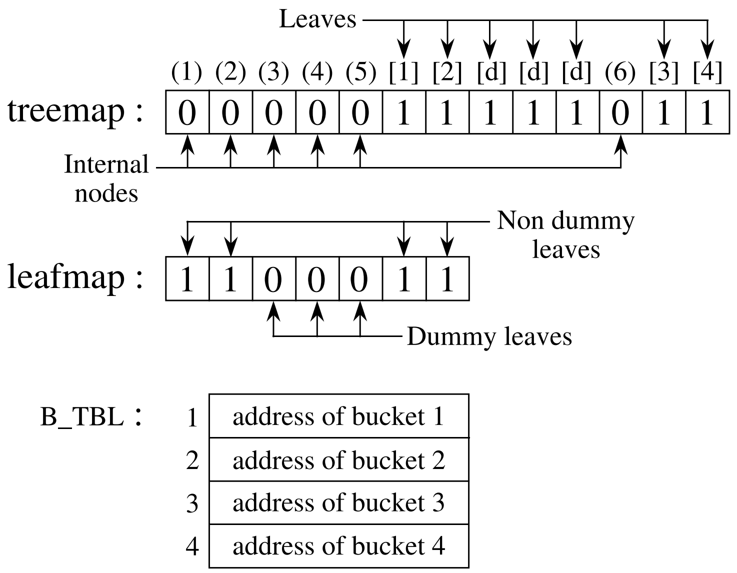


Figure 2: An example of the pre-order bit stream.

Step(S-2): {Skipping the left subtree}
If the bit of s_key pointed to by $keypos$ is a '1',
then $leafpos \leftarrow leafpos$+SKIP_COUNT();

Step(S-3): {Advance to the right subtree}
$keypos \leftarrow keypos+1$; $treepos \leftarrow treepos+1$;

Step(S-4): {Loop invariant until reaching the leaf}
If the bit of $treemap$ pointed to by $treepos$ is a '0', return to Step(S-2);

Step(S-5): {Verification of $leafmap$}
If the bit of $leafmap$ pointed to by $leafpos$ is a '0', FALSE is returned;

Step(S-6): {Verification of B_TBL}
$bucketnum \leftarrow$ FIND_BUCKET();
If the bucket indicated by $bucketnum$ contains the key, return TRUE, otherwise return FALSE;

Regarding the above algorithm, since a left subtree in $treemap$ is represented following the 0 bit of its parent node, when advancing to the left subtree, the Step(S-2) is not executed, however when advancing to the right subtree, the Step(S-2) to skip the left subtree is added. This skipping process utilizes the binary tree's property that the number of leaves is one more than the number of internal nodes in any binary subtree. Using this property, the function SKIP_COUNT() can search for the end position of the left subtree and get the number of leaves in the left subtree. Namely, this function advances $treepos$ until the number of 1 bits is one more than the number of 0 bits, and returns the number of 1 bits (leaves). Moreover, the value obtained by counting the number of 1 bits in $leafmap$ from the first bit to the one pointed to by $leafpos$ indicates which slot in B_TBL contains the required bucket address.

For example, to retrieve key="zoo" (s_key="11c") in Figure 2, the following steps are performed:

Step(S-1): *keypos*=*treepos*=*leafpos*=1; Since the first bit of *s_key* is a '1', the subtree whose root is node 2 is skipped by SKIP_COUNT().

Step(S-2): *leafpos*=*leafpos*+SKIP_COUNT()=6;

Step(S-3): *keypos*=2; *treepos*=11;

Step(S-4): Since the 11-th bit of *treemap* is a '0', return to Step(S-2);

Step(S-2‘4): Since the 2-th of *s_key* is a '1', the subtree whose root is node 6 is skipped; *leafpos*=*leafpos*+SKIP_COUNT()=7; *treepos*=13;

Step(S-5): Since the 7-th bit of *leafmap* is a '1', *B_TBL* is verified;

Step(S-6): Since key "zoo" is stored in the bucket 4, TRUE is returned;

3 Improvement by Using Hierarchical Structures

The BDS-tree represented by the pre-order bit stream is a very compact binary trie, however, the more keys are stored in the tree, the longer the bit strings (*treemap* and *leafmap*) are. As a result, the time-cost for each process is high. For example, as for the retrieval, the worst case is when search process is done toward the rightmost leaf in the BDS-tree as shown in Figure 3. In this case, if the rightmost leaf keeps the address of the bucket of the searching key, all bits in *treemap* (*leafmap* also) of the pre-order bit stream must be scanned. Similarly, in the case when an arbitrary key is inserted in the bucket corresponding to the leftmost leaf, suppose the bucket is divided and merge, all bits after the bit corresponding to the leftmost leaf in *treemap* of the pre-order bit stream have to be shifted. In this paper, the method to solve the problem stated above is proposed.

This method separates the BDS-tree into smaller BDS-trees of a certain depth. This depth is called the *separation depth*, and these small trees are called *separated trees*. These separated trees are numbered and connected by pointers. The BDS-tree separated in this way is called a Hierarchical Binary Digital Search tree (HBDS-tree). The HBDS-tree obtained based on the BDS-tree of Figure 4 -(a), with a separation depth of 2, is shown in Figure 4 -(b). In this case when rightmost leaf is searched, if we use the BDS-tree as shown in Figure 4 -(a), all internal nodes and leaves must be scanned in pre-order traversal. On the other hand, in the case of the HBDS-tree as shown in Figure 4 -(b), we can search the rightmost leaf by scanning all nodes and leaves of the only separated tree 1.

The algorithm to retrieve a key in the HBDS-tree uses the pre-order bit stream. The binary sequence H(k) of the key is divided into the following binary sequence:

$$H(k) = H_1(k)\ H_2(k)\ c\ H_j(k)\ c\ H_n(k)$$

Supposing that the separation depth is denoted by L, the lengths of $H_1(k)$‘$H_{n-1}(k)$ are L bits and the length of $H_n(k)$ is less than L bits. The HBDS-tree can be compressed into a very compact data structure named the pre-order bit stream as well as the BDS-tree. The pre-order bit stream is created and controlled for each of the separated trees. The pre-order bit stream that corresponds to the *i*-th separated tree in the

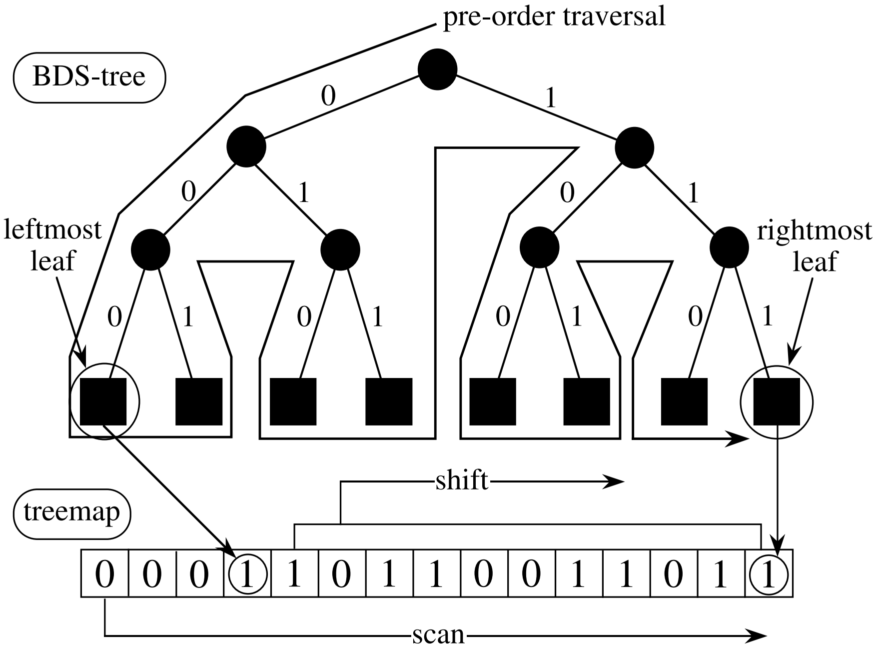


Figure 3: Retrieval of the BDS-tree in the worst-case.

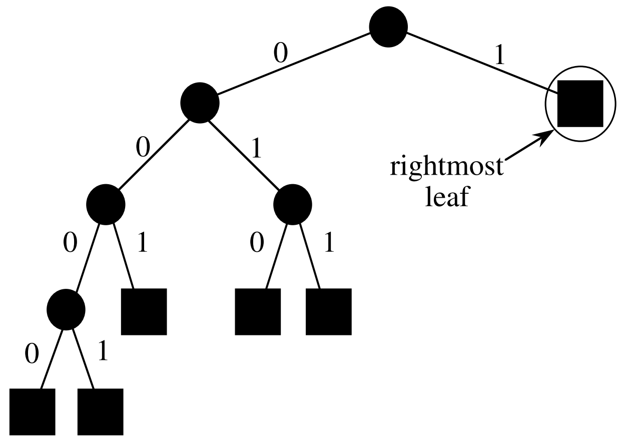
HBDS-tree consists of $treemap_i$, $leafmap_i$ and B_TBL_i, but the leaf which becomes the pointer to the next separated tree is regarded as a especial leaf and B_TBL_i contains the number of the next separated tree preceded by a minus sign in the slot corresponding to the leaf. The HBDS-tree obtained based on the BDS-tree of Figure 1, with a separated depth of 2, is shown in Figure 5, and the pre-order bit stream for the HBDS-tree of Figure 5 is shown in Figure 6, where, as can be seen above the $treemap$, the leaves which became the pointer to the separated tree are marked by "$\langle\rangle$". By using this improved method, each process can be sped up, because unnecessary scanning of the pre-order bit stream for each separated tree can be omitted.

The algorithm for retrieval of the HBDS-tree represented by the pre-order bit stream is shown below, where it uses the following variables:
i: The current separated tree number.
s_key: The key to be searched.
$keypos$: A pointer to the current position in s_key.
$treepos$: A pointer to the current position in $treemap_i$.
$leafpos$: A pointer to the current position in $leafmap_i$.
$bucketnum$: The corresponding bucket number.
Moreover, each of the functions performs the same process as the functions explained in Section 2 toward the i-th separated tree, when i is initialized with 1.

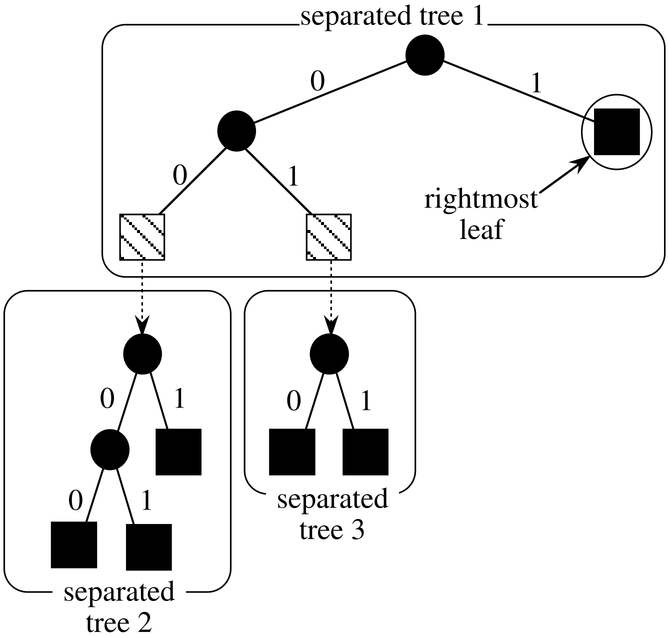
[An Algorithm to search in the HBDS-tree]
Input: s_key;
Output: If s_key can be found, then the output is TRUE, otherwise FALSE;

Step(S'-1)~Step(S'-5): The same procedures as the Step(S-1)'Step(S-5) are performed, however their $treemap$, $leafmap$ are changed into $treemap_i$, $leafmap_i$;

Step(S'-6): {Verification of $bucketnum$}



(a) **An example of the BDS-tree.**



(b) **An example of the HBDS-tree.**

Figure 4: Improvement of the BDS-tree by using hierarchical structures.

$bucketnum \leftarrow$ FIND_BUCKET(i);
If $bucketnum \leq 0$, proceed to Step(S'-7), otherwise proceed to Step(S'-8);

Step(S'-7): {Obtaining the separated tree number}
$i = -1 \times bucketnum$; Return to Step(S'-1);

Step(S'-8): {Verification of B_TBL}
If the bucket indicated by *bucketnum* contains the key, return TRUE, otherwise return FALSE;

For example, in the case of retrieval the key = zoo (s_key = "11c") in the pre-order bit stream of the HBDS-tree as shown in Figure 6, s_key can be retrieved in the HBDS-tree by using the pre-order bit stream of the only separated tree 1, so that the time-cost of retrieval becomes better than the case by using the BDS-tree's one.

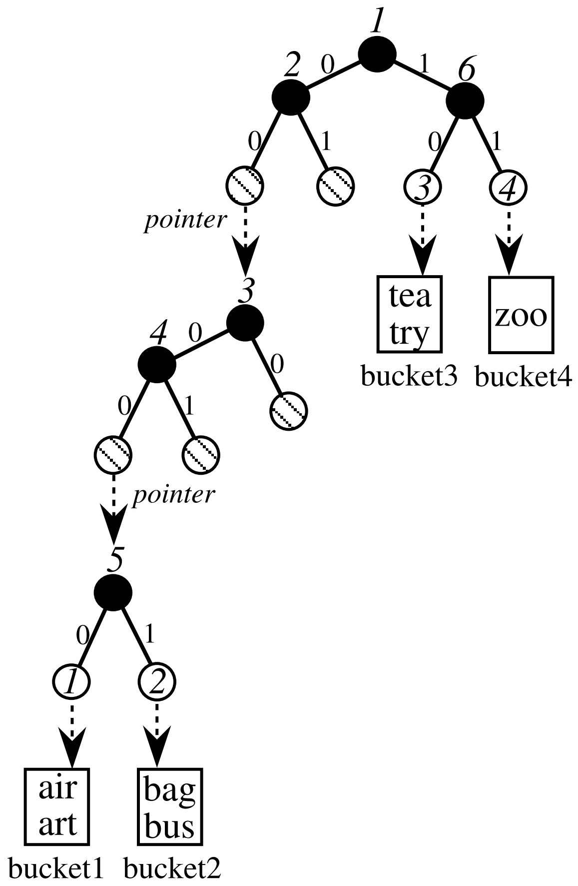


Figure 5: HBDS-tree based of the BDS-tree of Figure 1.

4 An Insertion Algorithm

The method for inserting the new key into the HBDS-tree is divided into the following three cases as well as the BDS-tree.
1) the required bucket is partially filled.
2) the required bucket is a dummy bucket.
3) the required bucket is full.
In this chapter, the third case, when the required bucket is full, that is, the method for dividing the full bucket into the new two buckets is explained. An explanation of the other cases is omitted, because they are very simple.

When there is an overflow in the required bucket, in the BDS-tree, the following processes are repeated until the overflow of the bucket does not happen. First, the corresponding leaf to the full bucket is changed into a tree which consists of a node and two dummy leaves. This tree is called a unit tree. Next, all the keys in the full bucket and an insertion key are distributed between the corresponding two buckets to dummy leaves of the unit tree. On the HBDS-tree, when the unit tree is made, a new separated tree must be created every time the depth of each separated tree exceeds the separation depth. As for the insertion process which uses the pre-order bit stream, a bit line "011", which represents the unit tree in $treemap$, and a bit line "00", which represents the two dummy leaves of the unit tree in $leafmap$, are inserted into $treemap$ and $leafmap$ respectively.

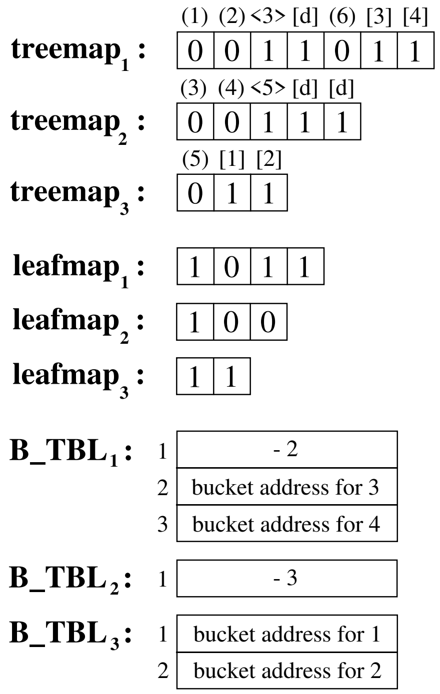


Figure 6: The pre-order bit stream for the HBDS-tree of Figure 5.

5 Evaluation

5.1 Theoretical Evaluation

In this section, the worst-case time complexities of each algorithm for the BDS-tree and HBDS-tree are theoretically analyzed. And the space complexities of each pre-order bit stream for the BDS-tree and HBDS-tree also are calculated. Let the tree structure to be analyzed be the complete tree. The following parameters are used:
n: The depth of the complete tree;
m: The separation depth;
α: The number of layers in the HBDS-tree. It is obtained by $[n/m]$, where $[n/m]$ indicates the minimum integer greater than or equal to n/m;

As for the time complexity, the worst-case time complexity for retrieval for the BDS-tree is $O(2^n)$, because the whole of the complete tree must be scanned. However, for the HBDS-tree it is $O(\alpha 2^m)$, since only α separate trees are scanned. Regarding the insertion and deletion, the worst case is when each process is done toward the leftmost bucket in the tree. In this case, suppose the bucket is divided and merged, the BDS-tree has a time complexity $O(2^n - n)$, because all bits after the bit corresponding to the bucket in the pre-order bit stream have to be shifted, however for the HBDS-tree it is $O(2^m - m)$, because the same operations are performed toward only one separated tree. Generally, for $n \ll 2^n$ and $m \ll 2^m$, the worst-case time complexity for insertion and deletion in the BDS-tree is $O(2^n)$ and for the HBDS-tree it is $O(2^m)$.

As for the space complexity, on the BDS-tree, the number of bits used for the *treemap* is equal to the total number of nodes (internal nodes and leaves) of the complete tree, that is, it is $2^{n+1} - 1$. And the leafmap needs 2^n bits which is the number of leaves in the complete tree. As for the sizes of the *treemap* and *leafmap*

for the HBDS-tree, they are calculated as shown below :
Number of bits required for *treemap*

$$= (\text{number of all nodes of the separated tree}) \times (\text{number of the separated trees within the complete tree})$$
$$= \sum_{k=0}^{m} 2^k \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = (2^{m+1} - 1)\frac{2^{m\alpha} - 1}{2^m - 1}$$
$$= \{2(2^m - 1) + 1\}\frac{2^{m\alpha} - 1}{2^m - 1} = (2^{m\alpha+1} - 2) + \frac{2^{m\alpha} - 1}{2^m - 1}$$
$$= (2^{n+1} - 1) + \frac{2^n - 1}{2^m - 1} - 1$$

Number of bits required for *leafmap*

$$= (\text{number of leaves of the separated tree}) \times (\text{number of the separated trees within the complete tree})$$
$$= 2^m \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = 2^m \frac{2^{m\alpha} - 1}{2^m - 1}$$
$$= (2^m - 1 + 1)\frac{2^{m\alpha} - 1}{2^m - 1} = (2^{m\alpha} - 1) + \frac{2^{m\alpha} - 1}{2^m - 1}$$
$$= 2^n + \frac{2^n - 1}{2^m - 1} - 1$$

From the above results, if the BDS-tree is separated, the storage requirement for both the treemap and the leafmap increases only $(2n - 1)/(2m - 1) - 1$ bits.

5.2 Experimental Evaluation

This method was written in about 2,000 lines of code in C, and implemented on a Sun Microsystems Sparc Station 2 (28 MIPS).

Key sets	Japanese nouns		English words	
Kinds of trees	BDS-tree	HBDS-tree	BDS-tree	HBDS-tree
Number of				
non_dummy leaves	6,002		6,159	
dummy leaves	3,649		8,411	
Internal nodes	9,650		14,569	
depth	82		70	
separated tree	2,060		2,940	
Time (Second)				
Registration	870	146	1875	164
Time (Milli-Second)				
Retrieval	8.68	0.48	11.26	0.56
Insertion	38.00	3.00	37.50	3.28
Storage (K-byte)				
treemap	2.41	2.67	3.64	4.00
leafmap	1.21	1.46	1.82	2.19
B_TBL	12.00	16.12	12.32	18.20

Table 1: Experimental results.

In order to observe the effect of this method, we compare the cost time of each process and storage requirement for the BDS-tree and the HBDS-tree. 50,000 nouns in Japanese and 50,000 English words with an average length of 6 and 9 bytes respectively are used as the key sets. Table 1 shows the experimental results for the each of key sets, where the separation depth is 5 and B_SIZE is 16. Retrieval time is the average time required for a key when all registered keys are searched and deleted, respectively. Insertion time is the average time required for a key when 1000 unregistered keys are added to the key set. Storage in Table 1 shows the memory required for the registration of the each key set.

From the experimental results, the retrieval in the HBDS-tree is 18‘20 times faster than in the BDS-tree, the insertion is 11‘13 times faster. Thus, it can be concluded that the time each of the processes requires is significantly less when using this method. As for the storage space required by the HBDS-tree, the sizes of $treemap$, $leafmap$ and B_TBL are 1.11, 1.21 and 1.34 times the size of the ones used by the BDS-tree. However, by nature, the pre-order bit stream is very compact in size, thus their sizes are good enough for practical applications. Moreover, for the BDS-tree and the HBDS-tree, both represented by the pre-order bit stream, the storage requirement to register one key is of 2.50 and 3.24 bits, respectively. Thus, these methods can be operated with more compact storage than the B-tree, B^+-tree, etc.

6 Conclusions

The Binary trie represented by the pre-order bit stream can search a key in order, however, the time-cost of each process becomes high for large key sets. So, the method for solving the above problem by separating the tree structure has been presented in this paper. The time and space efficiency of the proposed method is theoretically discussed, and the validity of this method has been supported by empirical observations. As future improvements, an efficient method to improve the space efficiency of the bucket should be designed.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, “Data Structures and Algorithms”, Reading, Mass.: Addison-Wesley, pp. 163–169, 1983.

[2] J. Aoe, “An Efficient Digital Search Algorithm by Using a Double-Array Structure”, *IEEE Trans. Software. Eng.*, Vol. 15, No. 9, pp. 1,066–1,077, 1989.

[3] J. Aoe, “Computer Algorithms-Key Search Strategies”, IEEE Comput. Society Press, 1991.

[4] G.H. Gonnet, “Handbook of Algorithms and Data Structures”, Addison-Wesley, Reading Mass. Ch. 3 (Searching Algorithms), pp. 25–147, 1984.

[5] W. D. Jonge, A.S. Tanenbaum and R.P. Reit, “Two Access Methods Using Compact Binary Trees”, *IEEE Trans. Software. Eng.*, Vol. 13, No. 7, pp. 799–809, 1987.

[6] M. Shishibori, S. Kiyohara and J. Aoe, “Improvement of Binary Digital Search (BDS)-Trees Using Hierarchical Structures”, in Japanese, *Trans. IEICE*, Vol. J79-D-I, No. 2, pp. 79–87, 1996.

A New Family of String Pattern Matching Algorithms

Bruce W. Watson, Richard E. Watson

Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada

e-mail: {watson, rwatson}@RibbitSoft.com

Abstract. Even though the field of pattern matching has been well studied, there are still many interesting algorithms to be discovered. In this paper, we present a new family of single keyword pattern matching algorithms. We begin by deriving a common ancestor algorithm, which naïvely solves the problem. Through a series of correctness preserving predicate strengthenings, and implementation choices, we derive efficient variants of this algorithm. This paper also presents one of the first algorithms which could be used to do a minimal number of match attempts within the input string (by maintaining as much information as possible from each match attempt).

Key words: single keyword pattern matching, shift distances, match attempts, reusing match information, predicate strengthening and weakening.

1 Introduction and related work

In this paper, we present a new family of algorithms solving the single keyword string pattern matching problem. This particular pattern matching problem can be described as follows: given an input string S and a keyword p, find all occurrences of p as a continuous substring of S. The field of string pattern matching is generally well-studied, however, it continues to yield new and exciting algorithms, as was seen in Watson's Ph.D. dissertation [Wats95], the recent book by Crochemore and Rytter [2], the more classic paper by Hume and Sunday [7] and book by Gonnet and Baeza-Yates [4]. In the dissertation [Wats95], a taxonomy of existing algorithms was presented, along with a number of new algorithms. Any given algorithm may have more than one possible derivation, leading to different classifications of the algorithm in a taxonomy[1]. Many of the new derivations can prove to be more than just an educational curiosity, possibly leading to interesting new families of algorithms. This paper presents one such family — with some new algorithms and also some alternative derivations of existing ones. While a few of the derivation steps are shared with the presentation

[1]This is precisely what happened with the Boyer-Moore type algorithms as presented in the dissertation [Wats95].

in [Wats95], this paper takes a substantially different approach overall and arrives at some completely new algorithms.

The algorithms presented in this paper can be extended to handle some more complex pattern matching problems, including multiple keyword pattern matching, regular pattern matching and multi-dimensional pattern matching.

Our derivation begins with a description of the problem, followed by a naïve first algorithm. We then make incremental (correctness preserving) improvements to these algorithms, eventually yielding efficient variants. Throughout the paper, we first precede each definition with some intuitive background. Before presenting the derivation, we give the mathematical preliminaries necessary to read this paper.

2 Mathematical preliminaries

While most of the mathematical notation and definitions used in this paper are is described in detail in [5], here we present some more specific notations. Indexing within strings begins at 0, as in the C and C++ programming languages. We use ranges of integers throughout the paper which are defined by (for integers i and j):

$$[i, j) = k | i \leq k < j$$

$$(i, j] = k | i < k \leq j$$

$$[i, j] = [i, j) \cup (i, j]$$

$$(i, j) = [i, j) \cap (i, j]$$

In addition, we define a *permutation* of a set of integers to be a bijective mapping of those integers onto themselves.

3 The problem and a first algorithm

Before giving the problem specification (in the form of a postcondition to the algorithms), we define a predicate which will make the postcondition and algorithms easier to read. Keyword p (with the restriction that $p \neq \varepsilon$, where ε is the empty string) is said to *match* at position j in input string S if $p = S_{j \cdots j+|p|-1}$; this is restated in the following predicate:

Definition 3.1 (Predicate *Matches***):** We define predicate *Matches* as

$$Matches(S, p, j) \equiv p = S_{j \cdots j+|p|-1}$$

□

The pattern matching problem requires us to compute the set of all matches of keyword p in input string S. We register the matches as the set O of all indices j (in S) such that $Matches(S, p, j)$ holds.

Definition 3.2 (Single keyword pattern matching problem): Given a common alphabet V, input string S, and pattern keyword p, the problem is defined using postcondition PM:

$$O = \{\, j \mid j \in [0, |S|) \wedge Matches(S, p, j)\,\}$$

Note that this postcondition implicitly depends upon S and p. □

We can now present a nondeterministic algorithm which keeps track of the set of possible indices (in S) at which a match might still be found (indices at which we have not yet checked for a match). This set is known as the *live zone*. Those indices not in the live zone are said to be in the *dead zone*. This give us our first algorithm (presented in the guarded command language of Dijkstra [3, 1]).

Algorithm 3.3:

$live, dead := [0, |S|), \emptyset;$
$O := \emptyset;$
$\{$ invariant: $live \cup dead = [0, |S|) \wedge live \cap dead = \emptyset$
$\qquad\qquad \wedge\, O = \{\, j \mid j \in dead \wedge Matches(S, p, j)\,\}\ \}$
do $live \neq \emptyset \rightarrow$
  let $j : j \in live;$
  $live, dead := live \setminus \{j\}, dead \cup \{j\};$
  if $Matches(S, p, j) \rightarrow O := O \cup \{j\}$
  $[\!]\ \neg Matches(S, p, j) \rightarrow$ **skip**
  fi
od$\{$ postcondition: PM $\}$

□

The invariant specifies that *live* and *dead* are disjoint and account for all indices in S; additionally, any match at an element of *dead* has already been registered. Thanks to this relationship between *live* and *dead*, we could have written the repetition condition $live \neq \emptyset$ as $dead \neq [0, |S|)$, and the j selection condition $j \in live$ as $j \notin dead$. It should be easy to see that the invariant and the termination condition of the repetition implies the postcondition — yielding a correct algorithm. Note that this algorithm is highly over-specified by keeping both variables *live* and *dead* to represent the live and dead zones, respectively. For efficiency, only one of these sets would normally be kept.

Some of the rightmost positions in S cannot possibly accommodate matches — no match can be found at any point $j \in [|S| - |p| + 1, |S|)$ since $|S_{j \cdots |S|-1}| \leq |S_{|S|-|p|+1 \cdots |S|-1}| < |p|$ (the match attempt begins too close to the end of S for p to fit). For this reason, we safely change the initializations of *live* and *dead* to

$$live, dead := [0, |S| - |p|], [|S| - |p| + 1, |S|)$$

In the next section, give a deterministic (more realistically implemented) version of the last algorithm.

4 A more deterministic algorithm

In the last algorithm, our comparison of p with $S_{j \cdots j+|p|-1}$ is embedded within the evaluation of predicate *Matches*. In this section, we make this comparison explicit. We begin by noting that $p = S_{j \cdots j+|p|-1}$ is equivalent to comparing the individual symbols p_k of p with the corresponding symbols S_{j+k} of S (for $k \in [0, |p|)$). In fact, we can consider the symbols in any order whatsoever. To determine the order in which they will be considered, we introduce *match orders*:

Definition 4.1 (Match order): We define a match order mo as a permutation on $[0, |p|)$. □

Using mo, we can restate our match predicate.

Property 4.2 (Predicate *Matches*): Predicate *Matches* is restated as

$$Matches(S, p, j) \equiv (\forall\ i : i \in [0, |p|) : p_{mo(i)} = S_{j+mo(i)})$$

□

This rendition of the predicate will be evaluated by a repetition which uses a new integer variable i to step from 0 to $|p| - 1$, comparing $p_{mo(i)}$ to the corresponding symbol of S. As i increases, the repetition has the following invariant:

$$(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})$$

and terminates as early as possible.

In the following algorithm, we use the match order mo, the new repetition and our previous optimization to the initializations of *dead* and *live*.

Algorithm 4.3:

$live, dead := [0, |S| - |p|], [|S| - |p| + 1, |S|);$
$O := \emptyset;$
$\{$ invariant: $live \cup dead = [0, |S|) \wedge live \cap dead = \emptyset$
$\wedge\ O = \{\ j \mid j \in dead \wedge Matches(S, p, j)\ \}\ \}$
do $live \neq \emptyset \rightarrow$
  let $j : j \in live;$
  $live, dead := live \setminus \{j\}, dead \cup \{j\};$
  $i := 0;$
  $\{$ invariant: $(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})\ \}$
  do $i < |p|$ **cand** $p_{mo(i)} = S_{j+mo(i)} \rightarrow$
    $i := i + 1$
  od;
  $\{$ postcondition: $(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})$
  $\wedge\ (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)})\ \}$
  if $i = |p| \rightarrow O := O \cup \{j\}$
  $[\!]\ \ i < |p| \rightarrow$ **skip**
  fi
od$\{$ postcondition: PM $\}$

□

The operator P **cand** Q appears in the guard of the inner loop of the above algorithm. This operator is similar to conjunction $P \wedge Q$ except that if the first conjunct evaluates to *false* then the second conjunct is not even evaluated. This proves to be a useful property in cases such as the loop guard since, if the first conjunct $(i < |p|)$ is *false* (hence $i >= |p|$, and indeed $i = |p|$), then the term $mo(i)$ appearing in the second conjunct is not even defined. Note that the implication within the second conjunct of the loop postcondition is derived from the loop guard, forcing the implication operator to be conditional as well (that is, if $i < |p|$ is determined to be *false*, then $p_{mo(i)} \neq S_{j+mo(i)}$ is not even evaluated).

As we will see in the next section, the particular choice for mo can make a difference in the performance of the algorithm. Some possible match orders include 'forward' (mo is the identity permutation) and 'reverse' ($mo(i) = |p| - i - 1$). The permutation chosen could even be devised according to some theoretical expectations or statistical analysis for a particular application. For instance, if p contained a subsequence of characters which are known to appear very rarely within the type of input string, then the permutation would be chosen in order to check for a match within that subsequence first (since this may result in discovering a mismatch sooner). This approach is standard fare, and is used to find fast variants of the Boyer-Moore algorithms (as described in [7]).

Yet another possibility which could prove interesting is that mo is chosen on-the-fly, that is, $mo(i)$ could be allowed to depend upon $mo(i-1)$, $mo(i-2)$, ..., $mo(0)$ and even upon other factors such as how much of the input string we have already processed. Such an choice of permutation would be highly specialized to a particular instance of this problem and we do not explore it any further in this paper. In the next section, we outline some precomputation on p which speeds up the algorithm tremendously but also depends upon the choice of mo, meaning that if we devised the permutation on-the-fly, we would be forced to perform the precomputation for each of the possible unique permutations that our algorithm could produce (a maximum of $|p|!$).

5 Reusing match information

On each iteration of the outer repetition, index j is chosen and eliminated from the live zone in the statement:

$$live, dead := live \setminus \{j\}, dead \cup \{j\}$$

The performance of the algorithm can be improved if we remove more than just j in some of the iterations. To do this, we can use some of the match information, such as i, which indicates how far through mo the match attempt proceeded before finding a mismatching symbol. The information most readily available is the postcondition of the inner repetition:

$$(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)})$$

We denote this postcondition by $Result(S, p, i, j)$. Since this postcondition holds, we may be able to deduce that certain indices in S cannot possibly be the site of a match.

It is such indices which we could also remove from the live zone. They are formally characterized as:

$$\{\, x \mid x \in [0, |S|) \wedge (Result(S,p,i,j) \Rightarrow \neg Matches(S,p,x)) \,\}$$

Determining this set at pattern matching time is inefficient and not easily implemented. We wish to derive a safe approximation of this set which can be precomputed, tabulated and indexed (at pattern matching time) by i. In order to precompute it, the approximation must be independent of j and S. We wish to find a strengthening of the range predicate since this will allow us to still remove a safe set of elements from set *live*, thanks to the property that, if $P \Rightarrow Q$ (P is a *strengthening* of Q, and Q is a *weakening* of P), then

$$\{\, x \mid P(x) \,\} \subseteq \{\, x \mid Q(x) \,\}$$

As a first step towards this approximation, we can normalize the ideal set (above), by subtracting j from each element. The resulting characterization will be more useful for precomputation reasons:

$$\{\, x \mid x \in [-j, |S| - j) \wedge (Result(S,p,i,j) \Rightarrow \neg Matches(S,p,j+x)) \,\}$$

Note that this still depends upon j, however, it will make some of the derivation steps shown shortly in Section 5.1 easier. Because those steps are rather detailed, they are presented in isolation. Condensed, the derivation appears as:

$$\begin{array}{ll}
 & (Result(S,p,i,j) \Rightarrow \neg Matches(S,p,j+x)) \\
\Leftarrow & \qquad \{\, \text{Section 5.1} \,\} \\
 & \neg((\forall\ k : k \in [0,i) \wedge mo(k) \in [x, |p|+x) : p_{mo(k)} = p_{mo(k)-x}) \\
 & \ \wedge (i < |p| \wedge mo(i) \in [x, |p|+x) \Rightarrow p_{mo(i)} \neq p_{mo(i)-x})) \\
\equiv & \qquad \{\, \text{define the predicate } Approximation(p,i,x) \,\} \\
 & Approximation(p,i,x)
\end{array}$$

Note that we define the predicate $Approximation(p,i,x)$ which depends only on p and i and hence can be precomputed and tabulated. It should be mentioned that this is one of several possible useful strengthenings which could be derived. We could even have used the strongest predicate, *false*, instead of $Approximation(p,i,x)$. This would yield the empty set, $\emptyset$, to be removed form *live* in addition to j (as in the previous algorithm).

We can derive a smaller range predicate of x for which we have to check if $Approximation(p,i,x)$ holds. Notice that choosing and x such that $[x, |p|+x) \cap [0, |p|) = \emptyset$ has two important consequences:

- The range of the quantification in first conjunct of $Approximation(p,i,x)$ is empty (hence this conjunct is *true*, by the definition of universal quantification with an empty range).

- The range condition of the second conjunct (the 'implicator') is *false* — hence the whole of the second conjunct is *true* since $false \Rightarrow P$ for all predicates P.

With this choice of x, we see that predicate $Approximation(p, i, x)$ always evaluates to *false*, in which case we need not even consider values of x such that $[x, |p| + x) \cap [0, |p|) = \emptyset$. This simplification can be seen in the following algorithm where we have solved the above range equation for x, yielding the restriction that $x \in [1 - |p|, |p| - 1)$. Intuitively we know that there must be such a range restriction since we can not possibly know from a current match attempt whether or not we will find a match of p in S more than $|p|$ symbols away.

Finally we have the following algorithm (in which we have added the additional update of *live* and *dead* below the inner repetition). Note that we introduce the set *nogood* to accumulate the indices for which $Approximation(p, i, x)$ holds. Also note that we renormalize the set *nogood* by adding j to each of its members and ensuring that it is within the valid range of indices, $[0, |S|)$.

Algorithm 5.1:

$live, dead := [0, |S| - |p|], [|S| - |p| + 1, |S|)$;
$O := \emptyset$;
{ invariant: $live \cup dead = [0, |S|) \wedge live \cap dead = \emptyset$
$\wedge\ O = \{ l \mid l \in dead \wedge Matches(S, p, l) \}$ }
do $live \neq \emptyset \rightarrow$
  let $j : j \in live$;
  $live, dead := live \setminus \{j\}, dead \cup \{j\}$;
  $i := 0$;
  { invariant: $(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})$ }
  do $i < |p|$ **cand** $p_{mo(i)} = S_{j+mo(i)} \rightarrow$
    $i := i + 1$
  od;
  { postcondition: $Result(S, p, i, j)$ }
  if $i = |p| \rightarrow O := O \cup \{j\}$
  $[\!]$ $i < |p| \rightarrow$ **skip**
  fi;
  $nogood := (\{ x \mid x \in [1 - |p|, |p| - 1) \wedge Approximation(p, i, x) \} + j)$
  $\cap\ [0, |S|)$;
  $live := live \setminus nogood$;
  $dead := dead \cup nogood$
od{ postcondition: PM }

□

5.1 Range predicate strengthening

Here, we present the derivation of a strengthening of the range predicate

$$Result(S, p, i, j) \Rightarrow \neg Matches(S, p, j + x)$$

Being more comfortable with weakening steps, we begin with the negation of part of the above range predicate, and proceed by weakening:

$$
\begin{array}{ll}
& \neg(\mathit{Result}(S,p,i,j) \Rightarrow \neg \mathit{Matches}(S,p,j+x)) \\
\equiv & \quad \{\text{ definition of } \Rightarrow \} \\
& \neg(\neg \mathit{Result}(S,p,i,j) \vee \neg \mathit{Matches}(S,p,j+x)) \\
\equiv & \quad \{\text{ De Morgan }\} \\
& \mathit{Result}(S,p,i,j) \wedge \mathit{Matches}(S,p,j+x)) \\
\equiv & \quad \{\text{ definition of } \mathit{Result} \text{ and } \mathit{Matches} \} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k : k \in [0,|p|) : p_{mo(k)} = S_{mo(k)+j+x}) \\
\equiv & \quad \{\text{ change range predicate in second quantification and definition of } mo \} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k : mo(k) \in [0,|p|) : p_{mo(k)} = S_{mo(k)+j+x}) \\
\Rightarrow & \quad \{\text{ change dummy } (mo(k') = mo(k)+x) \text{, restrict range }\} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k' : mo(k') - x \in [0,|p|) : p_{mo(k')-x} = S_{mo(k')+j}) \\
\equiv & \quad \{\text{ simplify range predicate of second quantification }\} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \wedge (i < |p| \Rightarrow p_{mo(i)} \neq S_{j+mo(i)}) \\
& \ \wedge (\forall\ k' : mo(k') \in [x,|p|+x) : p_{mo(k')-x} = S_{mo(k')+j}) \\
\Rightarrow & \quad \{\text{ one-point rule: second conjunct and second quantification }\} \\
& (\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)}) \\
& \ \wedge ((i < |p| \wedge mo(i) \in [x,|p|+x)) \Rightarrow p_{mo(i)} \neq p_{mo(i)-x}) \\
& \ \wedge (\forall\ k' : mo(k') \in [x,|p|+x) : p_{mo(k')-x} = S_{mo(k')+j}) \\
\Rightarrow & \quad \{\text{ combine two quantifications and remove dependency on } S \\
& \quad\ \ \text{and transitivity of } = \} \\
& (\forall\ k : k \in [0,i) \wedge mo(k) \in [x,|p|+x) : p_{mo(k)} = p_{mo(k)-x}) \\
& \ \wedge ((i < |p| \wedge mo(i) \in [x,|p|+x)) \Rightarrow p_{mo(i)} \neq p_{mo(i)-x})
\end{array}
$$

6 Choosing j from the live zone

In this section, we discuss strategies for choosing the index j (from the live zone) at which to make a match attempt. In the last algorithm, the way in which j is chosen from set *live* is nondeterministic. This leads to the situation that *live* (and, of course, *dead*) is fragmented, meaning that an implementation of the algorithm would have to maintain a set of indices for live. If we can ensure that *live* is contiguous, then an implementation would only need to keep track of the (one or two) boundary points between *live* and *dead*. There are several ways to do this, and we discuss some of them in the following subsections section. Each of these represents a particular *policy* to be used in the selection of j.

6.1 Minimal element

We could use the policy of always taking the minimal element of *live*. In that case, we can make some simplifications to the algorithm (which, in turn, improve the algorithm's performance):

- We need only store the minimal element of *live*, instead of sets *live* and *dead*. We use $\widehat{live}$ to denote the minimal element.
- The dead zone update could be modified as follows: we will have considered all of the positions to the left of j and so we can ignore the negative elements of the update set:

  $$\{\, x \mid x \in [1-|p|, 0) \wedge \mathit{Approximation}(p,i,x) \,\}$$

  Indeed, we can just add the maximal element (which is still contiguously in the update set and greater than j) of the update set to $\widehat{live}$ for the new version of our new update of *live* and *dead*.

Depending upon the choice of weakening, and the choice of match order, the above policy yields variants of the classical Boyer-Moore algorithm (see [Wats95, 2, 7]):

Algorithm 6.1:

$\widehat{live} := 0;$
$O := \emptyset;$
do $\widehat{live} \leq |S| - |p| \rightarrow$
  $j := \widehat{live};$
  $\widehat{live} := \widehat{live} + 1;$
  $i := 0;$
  $\{$ invariant: $(\forall\ k : k \in [0,i) : p_{mo(k)} = S_{j+mo(k)})\ \}$
  do $i < |p|$ **cand** $p_{mo(i)} = S_{j+mo(i)} \rightarrow$
    $i := i + 1$
  od;
  $\{$ postcondition: $\mathit{Result}(S,p,i,j)\ \}$
  if $i = |p| \rightarrow O := O \cup \{j\}$
  $[\!]$ $i < |p| \rightarrow$ **skip**
  fi;
  $\widehat{nogood} := (\mathbf{MAX}\ x : x \in [0, |p|-1)$
      $\wedge\ (\forall\ h : h \in [0,x] : \mathit{Approximation}(p,i,x)) : x);$
  $\widehat{live} := \widehat{live} + \widehat{nogood}$
od$\{$ postcondition: PM $\}$

□

6.2 Maximal element

We could always choose the maximal element of *live*. This would yield the dual of the previous algorithm.

6.3 Randomization

We could randomize the choice of j. Given the computational cost of most reasonable quality pseudo-random number generators, it is not clear yet that this would yield an interesting or efficient algorithm. It is conceivable that there exist instances of the problem which could benefit from randomly selected match attempts.

6.4 Recursion

We could also devise a recursive version of the algorithm as a procedure. This procedure receives a contiguous range of live indices (*live*) — initially consisting of the range $[0, |S| - |p|]$.

If the set it receives is empty, the procedure immediately returns. If the set is non-empty, j is chosen so that the resulting dead zone would appear reasonably close to the middle of the current live zone[2]. This ensures that we discard as little information as possible from the *nogood* index set. After the match attempt, the procedure recursively invokes itself twice, with the two reduced live zones on either side of the new dead zone. This yields the following procedure:

Algorithm 6.2:

---

proc $mat(S, p, live, dead) \rightarrow$
  { *live* is contiguous }
  if $live = \emptyset \rightarrow$ **skip**
  ▯ $live \neq \emptyset \rightarrow$
    $live\_low := (\textbf{MIN}\ k : k \in live : k)$;
    $live\_high := (\textbf{MAX}\ k : k \in live : k)$;
    $j := \lfloor (live\_low + live\_high - |p|)/2 \rfloor$;
    $i := 0$;
    { invariant: $(\forall\ k : k \in [0, i) : p_{mo(k)} = S_{j+mo(k)})$ }
    do $i < |p|$ **cand** $p_{mo(i)} = S_{j+mo(i)} \rightarrow$
      $i := i + 1$
    od;
    { postcondition: $Result(S, p, i, j)$ }
    if $i = |p| \rightarrow O := O \cup \{j\}$
    ▯ $i < |p| \rightarrow$ **skip**
    fi;
    $new\_dead := (\{ x \mid x \in [1 - |p|, |p| - 1) \wedge Approximation(p, i, x) \} + j)$
          $\cap [0, |S|)$;
    $dead := dead \cup new\_dead$;
    $mat(S, p, [live\_low, (\textbf{MIN}\ k : k \in new\_dead : k)), dead)$;
    $mat(S, p, ((\textbf{MAX}\ k : k \in new\_dead : k), live\_high], dead)$
  fi
corp

□

[2]The algorithm given in this section makes a simple approximation by taking the middle of the live zone it receives, and subtracting $\lfloor |p|/2 \rfloor$.

This procedure is used in the algorithm:

Algorithm 6.3:

$O := \emptyset;$
$mat(S, p, [0, |S| - |p|], [|S| - |p| + 1, |S|))$
$\{$ postcondition: PM $\}$

$\square$

Naturally, for efficiency reasons, the set *live* can be represented by its minimal and maximal elements (since it is contiguous).

7 Further work

The family of algorithms presented in this paper can easily be extended to multiple pattern matching and to regular pattern matching (using regular expressions or regular grammars). In each of these cases, various strengthenings of the update predicate could be explored and specialized methods for choosing the index of the next match attempt determined.

Another branch in this family tree of algorithms could be derived by removing the conjunct $p_{mo(i)} = S_{j+mo(i)}$ from the guard of the inner repetition (that is, do not terminate the match attempt as soon as we encounter a mismatch). This would allow us to accumulate more mismatch information and possibly provide a weaker strengthening than $Approximation(p, i, x)$ (and hence a larger set *nogood*). It is not yet clear that this would lead to an interesting family of algorithms.

Few of the algorithms presented here have been implemented in practice. Some of the algorithms presented here can be manipulated to yield the well-known Boyer-Moore variants, and we can therefore speculate that their running time is excellent, based upon the results presented in [Wats95]. It would be interesting to see how the new algorithms perform against the existing variants.

8 Conclusions

We have shown that there are still many interesting algorithms to be derived within the field of single keyword pattern matching. The correctness preserving derivation of am entirely new family of such algorithms demonstrates the use of formal methods and the use of predicates, invariants, postconditions and preconditions. It is unlikely that such a family of algorithms could have be devised without the use of formal methods.

Historically, keyword pattern matching algorithms have restricted themselves to processing the input string from left to right, thus discarding half of the useful information which can be determined from previous match attempts. As a new starting point for pattern matching algorithms, this paper proposes pattern matching in the more general manner of making match attempts in a less restricting order within the input string. With the advent of both hardware and software which enable near-constant-time lookup of a random character in a file stream (using memory mapped files, as are available in most newer operating systems), such algorithms will prove useful for typical single keyword pattern matching applications (ones which have a finite input string which can be randomly accessed).

The derivation also yielded a recursive algorithm which appears to be particularly efficient. The algorithm has been implemented, and benchmarking results will be presented in the final paper, comparing the algorithm to the other extensively benchmarked algorithms in [7, Wats95].

9 Acknowledgements

We would like to thank Nanette Saes and Mervin Watson for proofreading this paper, Ricardo Baeza-Yates for serving as a sounding board, and Ribbit Software Systems Inc. for allowing us to pursue some of our pure research interests.

References

[1] Cohen, E. *Programming in the 1990s,* (Springer-Verlag, New York, NY, 1990).

[2] Crochemore, M. and W. Rytter. *Text Algorithms,* (Oxford University Press, Oxford, England, 1994).

[3] Dijkstra, E.W. *A discipline of programming,* (Prentice Hall, Englewood Cliffs, NJ, 1976).

[4] Gonnet, G.H. and R. Baeza-Yates. *Handbook of Algorithms and Data Structures (In Pascal and C),* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[5] Gries, D. and F.B. Schneider. *A Logical Approach to Discrete Math,* (Springer-Verlag, New York, NT, 1993).

[6] Gries, D. *The Science of Programming,* (Springer-Verlag, New York, NY, 1981).

[7] Hume, S.C. and D. Sunday. "Fast string searching," *Software — Practice & Experience,* **21**(11) 1221–1248.

[8] Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms,* Ph.D dissertation, Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, September 1995, ISBN 90-386-0396-7.

6D Classification of Pattern Matching Problems[1]

Bořivoj Melichar, Jan Holub

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo náměstí 13
121 35 Prague 2
Czech Republic

e-mail: {melichar, holub}@cs.felk.cvut.cz

Abstract. We present our unified view to pattern matching problems and their solutions. We classify pattern matching problems by using six criteria and therefore we can locate them into six-dimensional space. We also show basic model of nondeterministic finite automaton that can be used for constructing models for all pattern matching problems.

Key words: string matching, sequence matching, classification, finite automata

1 Introduction

Pattern matching (string and sequence matching) appears as a very important component of many applications, including text editing, word processing, data retrieval, symbol manipulation, alignment in genetics, etc. This problem has been extensively studied since beginning of seventies.

The exact string matching is based on the two historical papers by Knuth, Morris and Pratt [KMP77] and by Boyer and Moore [BM77]. Matching of sequences started by Chvátal, Klainer and Knuth [CKK72]. They designed an algorithm for the longest common subsequence problem. Approximate string matching is based on well known paper by Wagner and Fischer [WF74]. Multiple string matching was originated by Aho and Corasick [AC75].

Since this first algorithms, many different problems of pattern matching were studied and many excellent ideas are included in existing text algorithms [CR94].

All one-dimensional pattern matching problems are sequential problems and therefore it is possible to solve them using finite automata. Below we discuss construction of finite automata for many pattern matching problems. These automata are mostly nondeterministic. There are three ways how these automata can be used:

1. To serve as a model of algorithms for solving of different problems.

2. To simulate the nondeterministic automaton in a deterministic way. Some of known pattern matching algorithms use this approach.

[1]This work was supported by grant FRVŠ 0892/97.

3. To construct an equivalent deterministic finite automaton. This approach may lead to the high space complexity in some cases.

The use of finite automata for the modelling of pattern matching algorithms means, that there is a formal method introduced to this part of computer science. It is well known from other areas (e.g. language theory) that introduction of formal approach has positive consequences. The main advantage is that it is possible to describe all problems using an unified view. This leads to the possibility to transfer know-how from another, well developed area, to compare different solutions, to find limitations, etc.

One of the consequences of the introduction of finite automata formalism is:

The possibility of the construction of finite automata for all problems in question shows, that there exist algorithms for all pattern matching problems having the linear time complexity. The space complexity is different for different problems. Existing upper bounds on space complexity of some problems are pessimistic.

Evaluation of some existing algorithms as simulators of nondeterministic finite automata will lead to understanding how some classes of automata can be simulated. This knowledge may serve for improvement of other algorithms and for the design of new ones.

2 Pattern matching problems

2.1 Basic notions and notations

Some basic notions will be used in subsequent sections. This section collects definitions of them. The notion pattern matching is used for string matching and sequence matching.

"Don't care" symbol is a special universal symbol Ø that matches any other symbol including itself.

Definition 1 (Basic pattern matching problems)
Given a text string $T = t_1 t_2 \cdots t_n$ and a pattern $P = p_1 p_2 \cdots p_m$. Then we may define:

1. String matching: verify if string P is a substring of text T.

2. Sequence matching: verify if sequence P is a subsequence of text T.

3. Subpattern matching: verify if a subpattern of P (substring or subsequence) occurs in text T.

4. Approximate pattern matching: verify if pattern P occurs in the text T so that the distance $D(P, X) \leq k$ for given $k < m$, where $X = t_i \cdots t_j$ is a part of text T.

5. Pattern matching with "don't care" symbols: verify if pattern P containing "don't care" symbols occurs in text T.

Definition 2 (Matching a sequence of patterns)
Given a text string $T = t_1 t_2 \cdots t_n$ and a sequence of patterns (string and sequences)

$P_1, P_2, \cdots, P_s$. Matching of sequence of patterns $P_1, P_2, \cdots, P_s$ is a verification whether an occurrence of pattern P_i in text T is followed by an occurrence of P_{i+1}, $1 \leq i < s$.

Definitions 1 and 2 define pattern matching problems as a decision problems, because the output is a Boolean value. A modified version of these problems consists in searching for the first, the last, or all occurrences of pattern and moreover the result may be the set of positions of the pattern in the text. Instead of just one pattern, one can consider a finite or infinite set of patterns.

Definition 3 (Distance of patterns)
Three variants of distances between two patterns X and Y are defined as minimal number of editing operations:

1. *replace* (Hamming distance, R-distance),
2. *delete, insert* and *replace* (Levenshtein distance, DIR-distance),
3. *delete, insert, replace* and *transpose* (generalized Levenshtein distance, $DIRT$-distance),

needed to convert pattern X into pattern Y.

The Hamming distance is a metrics on the set of string of equal length. The Levenshtein and the generalized Levenshtein distances are metrics on the set of strings not necessarily of equal length.

2.2 Classification of pattern matching problems

One-dimensional pattern matching problems for a finite size alphabet can be classified according to several criteria. We will use six criteria for classification leading to six-dimensional space in which one point corresponds to particular pattern matching problem.

Let us make a list of all dimensions including possible "values" in each dimension:

1. Nature of the pattern: string, sequence.
2. Integrity of the pattern: full pattern, subpattern.
3. Number of patterns: one, finite number, infinite number.
4. The way of matching: exact, approximate matching with Hamming distance (R-matching), approximate matching with Levenshtein distance (DIR-matching), approximate matching with generalized Levenshtein distance ($DIRT$-matching).
5. Importance of symbols in pattern: take care of all symbols, don't care of some symbols.
6. Number of instances of pattern: one, finite sequence.

The above classification is visualized in Figure 1. If we count the number of possible pattern matching problems, we obtain $N = 2 * 2 * 3 * 4 * 2 * 2 = 192$.

In order to make references to particular pattern matching problem easy, we will use abbreviations for all problems. These abbreviations are summarized in Table 1.

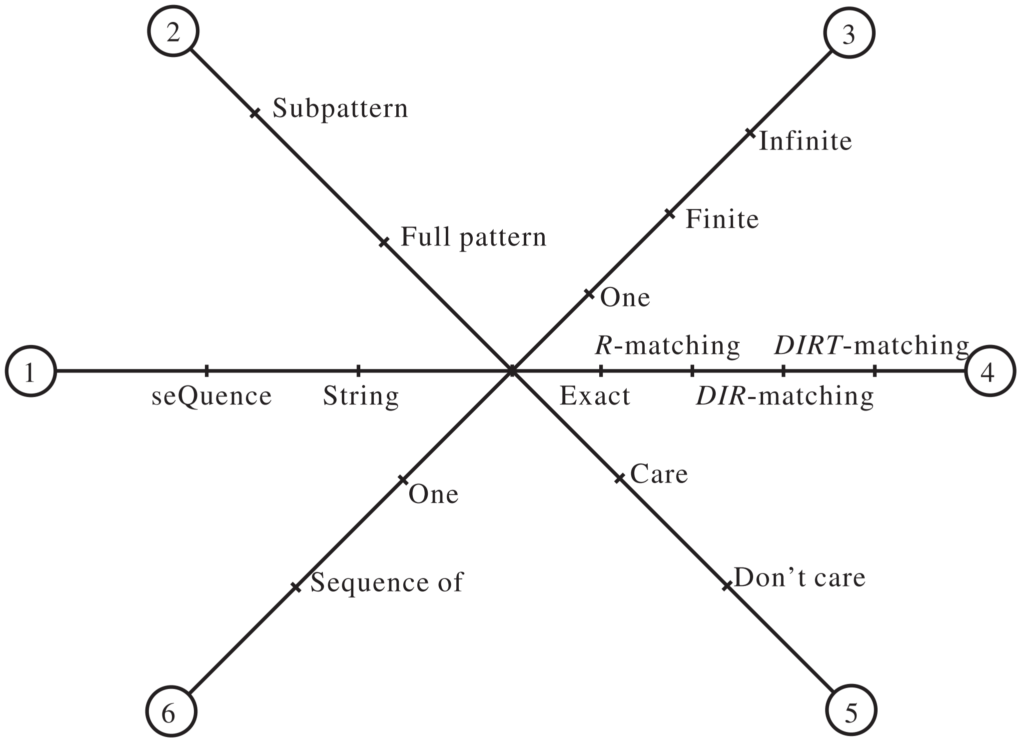


Figure 1: Classification of pattern matching problems.

Dimension	1	2	3	4	5	6
	S Q	F S	O F I	E R D G	C D	O S

Table 1: Abbreviations of pattern matching problems.

Using this method, we can, for example, refer to exact string matching of one string as $SFOECO$ problem.

Instead of single pattern matching problem we will use the notion of family of pattern matching problems. In this case we will use symbol '?' instead of particular letter. For example SFO??? is the family of all problems concerning one full string matching.

Each of pattern matching problem can have several instances. For example, $SFOECO$ problem can have the following instances:

1. verify whether given string occurs in text or not,

2. find the first occurrence of given string,

3. find the number of all occurrences of given string,

4. find all occurrences of given string and where they are.

If we take into account all possible instances, the number of pattern matching problems is further growing.

2.3 Pattern matching algorithms

Many algorithms for pattern matching problems were designed using ad hoc approach. But pattern matching problems are sequential problems and therefore it is possible to solve them using finite automata. There is possible to use systematic approach and to create a model of algorithm for each pattern matching problem. This model is nondeterministic finite automaton (*NFA*) in all cases.

We can convert an *NFA* to a deterministic finite automaton (*DFA*) and run it using the text as an input. If we suppose a finite size alphabet, then the running time of *DFA* is $\mathcal{O}(n)$, where n is the length of text.

It is well known, that the *NFA* to *DFA* conversion requires at most $\mathcal{O}(2^m)$ time, where m is the number of states of the *NFA*. The resulting *DFA* may have at most $\mathcal{O}(2^m)$ states.

Latest investigation shows, that this bound does not take place in the area of pattern matching problems [Me95], [Me96]. The bound of number of states of *DFA* is much lower and therefore this approach may have practical applications. Instead of conversion of *NFA* to *DFA*, we can simulate the *NFA* in a deterministic way. Some of known pattern matching algorithms use this approach. The problem of this approach is a high time complexity while the space complexity is low. In case of approximate string matching, "Shift-Or" based algorithms [BG92], [WM92] use this approach as it is shown in [Me95], [Me96].

3 Models of pattern matching algorithms

We will show, in this section, basic model of pattern matching algorithms. Moreover, we will show how to construct some models for more complicated problems using models of simple problems.

3.1 Exact string and sequence matching

In Fig. 1 the basic model of pattern matching algorithms is represented by the circle that meets all the axes in the points which are the closest to the junction of the axes. This model is the model for exact string matching ($SFOECO$ problem). For pattern $P = p_1p_2p_3p_4$ this model is shown in Fig. 2. The $SFOECO$ automaton has $m+1$ states for the pattern of the length m. This *NFA* can be transformed to *DFA* which has the same number of states as its nondeterministic version. The transformation can be performed in time $\mathcal{O}(m)$.

If we add loops labeled by mismatching characters into states, from which there leads at least one edge, in the model for string matching we obtain a corresponding model for sequence matching. The model of algorithm for exact sequence matching ($QFOECO$ problem) for pattern $P = p_1p_2p_3p_4$ is shown in Fig. 3. Character $\overline{p}$ represents any character mismatching character p. The $QFOECO$ automaton has $m+1$ states for the pattern of length m.

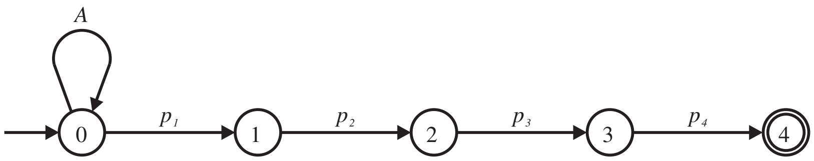


Figure 2: *NFA* for exact string matching ($SFOECO$ automaton)

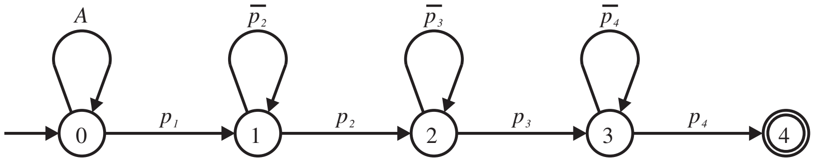


Figure 3: *NFA* for exact sequence matching ($QFOECO$ automaton).

3.2 Substring and subsequence matching

The model of algorithm for exact substring matching ($SSOECO$ problem) for pattern $P = p_1p_2p_3p_4$ is shown in Fig. 4. We can see that this automaton has been created by connecting m $SFOECO$ automata. The $SSOECO$ automaton has $(m+1)+m+(m-1)+\cdots+2 = \frac{m(m+3)}{2}$ states and is called an *initial ε-treelis*.

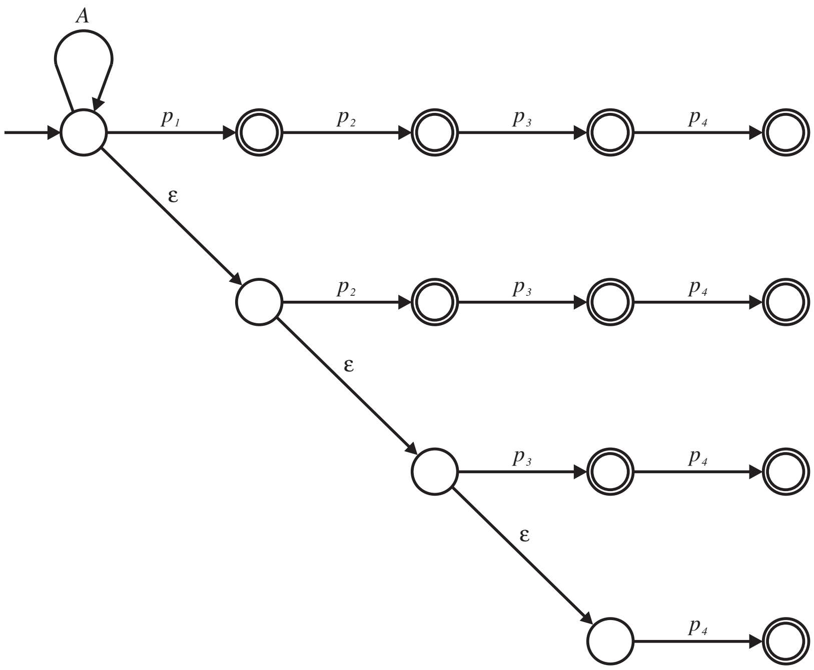


Figure 4: *NFA* for exact substring matching ($SSOECO$ automaton).

The model of algorithm for exact subsequence matching ($QSOECO$ problem) is similar as for exact substring matching. We get this model from $SSOECO$ model by adding loops for mismatching characters and ε-transitions into all states from which just one transition leads. Each ε-transition leads from state q_i to the state q_j such that from state, which is just under state q_i, a matching transition leads to state q_j. This automaton can be also constructed by connecting m $QFOECO$ automata. The $QSOECO$ automaton has $\frac{m(m+3)}{2}$ states and is called *ε-treelis*.

3.3 Approximate string matching

We will discuss three variants of approximate string matching corresponding to three definitions of distances between strings: Hamming distance, Levenshtein distance, and generalized Levenshtein distance.

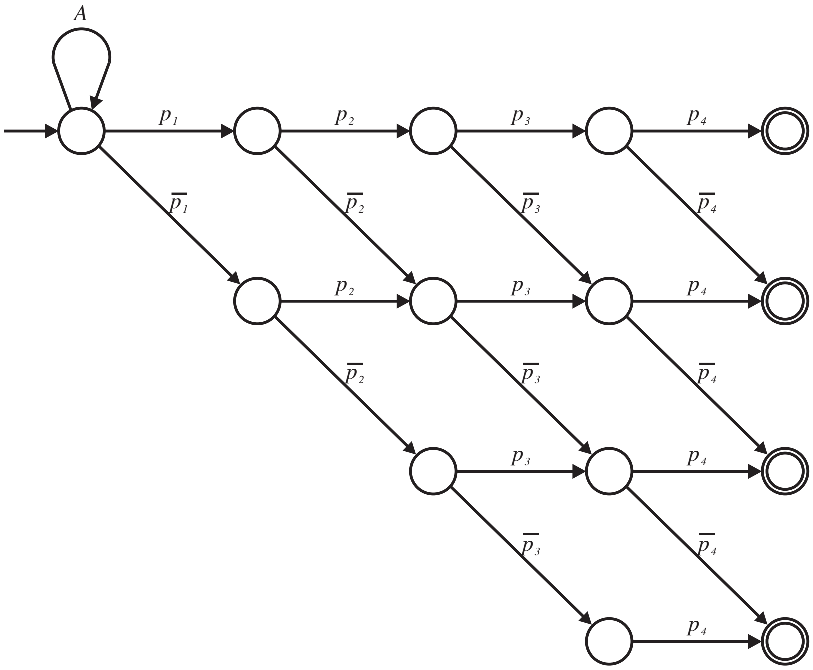


Figure 5: *NFA* for string R-matching ($SFORCO$ automaton).

Hamming distance Let us note, that Hamming distance between strings x and y is equal to the minimal number of editing operations *replace* which are necessary to convert string x into string y. Therefore this type of string matching is called string R-matching. The model of algorithm for string R-matching ($SFORCO$ problem) was presented in [Me95] and in Fig. 5 it is shown for string $P = p_1p_2p_3p_4$ and Hamming distance $k = 3.$. This automaton has been created by connecting $k+1$ $SFOECO$ automata by edges that represent editing operation *replace*. The $SFORCO$ automaton has $(m+1)+m+(m-1)+\cdots+(m-k+1) = (k+1)(m+1-\frac{k}{2})$ states. This automaton is called $R - treelis$.

Levenshtein distance Let us note, that Levenshtein distance between strings x and y is equal to the minimal number of editing operations *delete*, *insert* and *replace* which are necessary to convert string x into string y. Therefore this type of string matching is called string DIR-matching. The model of algorithm for string DIR-matching ($SFODCO$ problem) was presented in [Me96], [Ho96] and in Fig. 6 it is shown for string $P = p_1p_2p_3p_4$. It has been created from $SFORCO$ model by adding edges representing editing operations *insert* and *delete*.

Generalized Levenshtein distance Let us note, that generalized Levenshtein distance between strings x and y is equal to the minimal number of editing operations *delete*, *insert*, *replace* and *transpose* which are necessary to convert string x into string y. Therefore this type of string matching is called string $DIRT$-matching. The model of algorithm for string $DIRT$-matching ($SFOGCO$ problem) for string $P = p_1p_2p_3p_4$

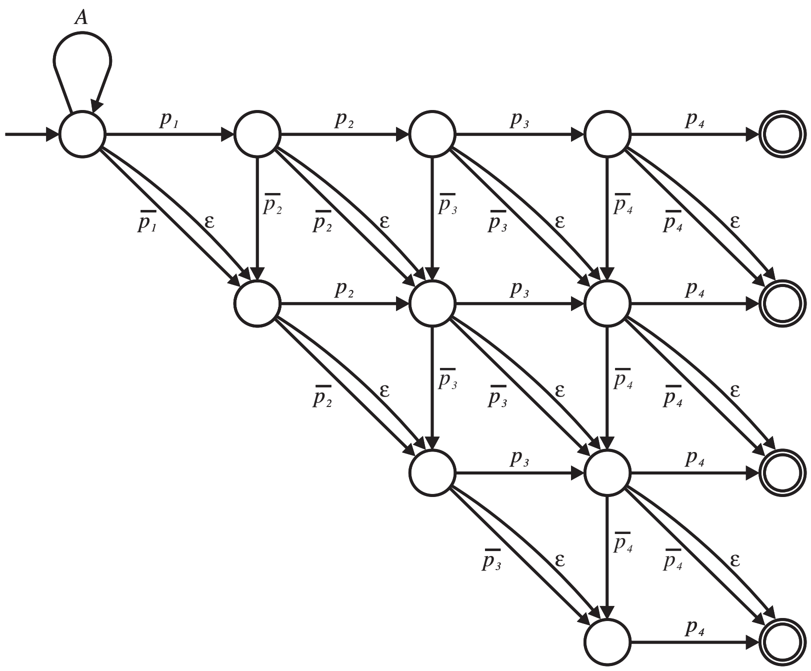


Figure 6: *NFA* for string *DIR*-matching (*SFODCO* automaton).

is shown in Fig. 7. It has been constructed from *SFODCO* model by adding states for editing operation *transpose* and corresponding edges.

Conclusion

We have presented unified view to pattern matching. We have also shown the basic model for pattern matching and several methods of constructing other models. These models can be very useful in designing new methods or improving existing methods for pattern matching. Since for each pattern matching problem there exists *NFA* there exists algorithm running in linear time for each such problem.

References

[AC75] Aho, A. V., Corasick, M. J.: Efficient String Matching: An Aid to Bibliographic Search. CACM, Vol. 18, No. 6, pp. 333–340, 1975.

[BG92] Baeza-Yates, R., Gonnet, G. H.: A New Approach to Text Searching. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 74–82.

[BM77] Boyer, R. S., Moore, J. S.: A Fast String Searching Algorithm. Commun. ACM, Vol. 20, No. 10, October 1977, pp. 762–772.

[CKK72] Chvátal, V., Klarner, D. A., Knuth, D. E.: Selected Combinatorial Research Problems. STAN-CS-72-292, Stanford University, June 1972, 26.

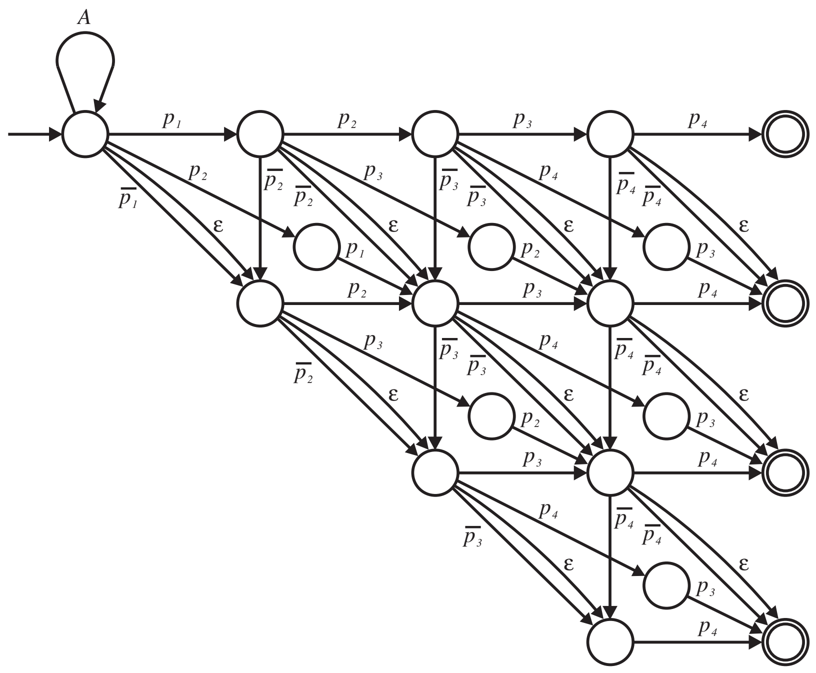


Figure 7: *NFA* for string *DIRT*-matching (*SFOGCO* automaton).

[CR94] Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press, New York 1994, p. 414.

[Ho96] Holub, J.: Reduced Nondeterministic Finite Automata for Approximate String Matching. Proceedings of the Prague Stringology Club Workshop '96, Czech Technical University, August 1996, pp. 19–27.

[KMP77] Knuth, D. E., Morris, J. H., Pratt, V. R.: Fast Pattern Matching in Strings, SIAM J. Comput., Vol. 6, No. 2, June 1977, pp. 322–350.

[Me95] Melichar, B.: Approximate String Matching by Finite Automata. Computer Analysis of Images and Patterns. LNCS 970, Springer 1995, pp. 342–349.

[Me96] Melichar, B.: String Matching with k Differences by Finite Automata. Proceedings of the 13^{th} ICPR, Vol. II, August 1996, pp. 256–260.

[WF74] Wagner, R. A., Fisher, M. J.: The String-to-String Correction Problem. Journal of ACM, January 1974, Vol. 21, No. 1, pp. 168–173.

[WM92] Wu, S., Manber, U.: Fast Text Searching Allowing Errors. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 83–91.

A Boyer-Moore (or Watson-Watson) Type Algorithm for Regular Tree Pattern Matching

Bruce W. Watson

Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada

e-mail: watson@RibbitSoft.com

Abstract. In this paper, I outline a new algorithm for regular tree pattern matching. The Boyer-Moore family of string pattern matching algorithms are considered to be among the most efficient. The Boyer-Moore idea of a shift distance was generalized by Commentz-Walter for multiple keywords, and generalizations for regular expressions have also been found. The existence of a further generalization to tree pattern matching was first mentioned in the statements accompanying my dissertation, [Wats95].

Key words: tree pattern matching, tree parsing, code selection, Boyer-Moore algorithms, shift distances.

1 Introduction

The most popular exact pattern matching algorithms[1] (for strings or trees) can be classified into one of two families: the Knuth-Morris-Pratt (KMP) or Boyer-Moore (BM) families.

Interest in Boyer-Moore type algorithms is driven by the fact that they are frequently much faster (in practice) than their KMP counterparts. For a discussion of this phenomenon, see [Wats95]. Since a KMP type algorithm for regular tree pattern matching was presented in [1], the missing piece has been a BM type algorithm for tree pattern matching. This algorithm is an extension (to trees) of one of the algorithms presented at the Prague Workshop in 1996 [4]; it is also related to the algorithm presented at the European Symposium on Algorithms in 1996 [3]. For more background material, consult the links on my homepage at `http://www.RibbitSoft.com/research/watson/index.html`

Instead of providing a set of formal definitions, we introduce most of the concepts using examples.

[1] As opposed to *approximate* pattern matching algorithms.

2 The problem

For the regular tree pattern matching problem, we consider node labeled trees (the labels are taken from a fixed alphabet). All nodes with a particular label have a fixed arity (number of children). We will write all of our trees in a linear (prefix) form, instead of drawing pictures; for example:

$$+(a, *(b, a))$$

Each of the nodes has an associated depth, with the depth of the root being 0.

A tree grammar is a finite set of productions, with nonterminals (written as uppercase letters, as opposed to regular node labels which are written in lowercase letters or as mathematical operators) on the left and tree templates on the right. Nonterminals are permitted to appear at the leaves of the tree templates. For example, each of the following lines is a production:

$$\begin{array}{lcl} A & \longrightarrow & +(B, B) \\ B & \longrightarrow & a \\ B & \longrightarrow & *(C, a) \\ C & \longrightarrow & b \\ C & \longrightarrow & *(b, B) \end{array}$$

The productions match at the input tree nodes in the intuitive way. In our sample input tree ($+(a, *(b, a))$), we have the following matched patterns:

- The left a is matched by $B \longrightarrow a$.
- The right a is matched by $B \longrightarrow a$.
- The b node is matched by $C \longrightarrow b$.
- The $*$ node is matched by $B \longrightarrow *(C, a)$ and by $D \longrightarrow *(b, B)$.
- The $+$ node is matched by $A \longrightarrow +(B, B)$.

In the next section, we will subdivide the pattern matching problem into a smaller problem which can be solved more readily.

3 Subproblems

One way of reducing the pattern matching problem to a simpler one, is to encode the trees as strings. We do this using so-called *path strings*. In this scheme, the tree is represented as a set of strings (there is one string for each leaf in the tree). Each string consists of alternating node labels and child numbers. For example, our input tree $+(a, *(b, a))$ is represented by $+1a$, $+2*1b$, and $+2*2a$.

In a similar manner, we encode each of the right sides of the productions as a set of path strings. The only difference is that we omit the nonterminals. For example, production $A \longrightarrow +(B, B)$ is represented by the two path strings $+1B$ and $+2B$, from which we drop the nonterminals to get $+1$ and $+2$. The example set of right sides is encoded as $+1$, $+2$, a, $*1$, $*2a$, b, $*1b$, and $*2$. These path strings can then be mapped back to their corresponding production right sides. These *pattern path* *strings* will be used in a reduced problem. Note that the pattern path strings will always begin with a node label.

Given this encoding, we will only concern ourselves with finding matches of the pattern path strings in the set of strings representing the input tree. The matching tree productions can then be easily reconstructed — this is not considered further here. In our example set of input path strings, we have the following pattern path string matches:

- $+1$ and $+2$ match at the root.
- a matches at the left and the right a nodes.
- $*1$, $*2$, $*1b$, and $*2a$ match at the $*$ node.
- b matches at the b node.

From this information, we can then piece together the tree matches. Note that, in effect, we are making use of multiple keyword pattern matching with the pattern path strings as the keywords. To solve this problem, we could use the Commentz-Walter algorithm (among others) [Wats95, Section 4.4].

4 Solving the reduced problem

We begin by presenting a brute-force (naïve) algorithm, solving our simplest subproblem. In presenting the algorithm, we will assume (as in the string pattern matching algorithms presented in my dissertation) the following:

- We use a forward trie τ (constructed from the pattern path strings) for the actual pattern matching. The symbol $\perp$ is used to indicate when the trie takes an undefined value.
- We assume that the start state for the trie is named q_0.
- There is a special procedure RM (for 'register matches') which is used to register matches at nodes in the tree. Precisely how it registers the matches is not relevant.

The mainline of the algorithm is:

$lev := (\mathbf{MAX}\ n : n \in nodes : n.level);$
do $lev \geq 0 \longrightarrow$
  for $n : n \in nodes \land n.level = lev \longrightarrow$
    $AM(q_0, n)$
  rof;
  $lev := lev - 1$
od

This algorithm simply traverses the tree from the bottom up, using procedure AM (for 'attempt match') to check for matches and RM to register the matches. The procedure AM is given as:

```
proc AM(q, n) is
    RM(q, n);
    if τ(q, n.label) ≠ ⊥ then
        q := τ(q, n.label);
        RM(q, n);
        for i ∈ [1, n.arity] ⟶
            if τ(q, i) ≠ ⊥ then
                AM(τ(q, i), n.child(i))
            fi
        rof
    fi
corp
```

This procedure uses the trie and traverses the input tree (starting at node n) top-down to find matches. Note that it is recursive.

As with the other BM type algorithms, we wish to make shifts of more than one level (in the tree) in the mainline program. The shift will be computed in a manner similar to that in the Commentz-Walter family of algorithms — since we are using multiple keyword pattern matching.

Since procedure AM tries a number of possible paths rooted at a node n, there will be a number of potential contributing shifts. In order to make a *safe* shift, we will have to use the smallest of these contributing shifts.

We will use a novel method of implementing the actual shift: if a shift of distance k is required after a match attempt at node n, we will store $n.level - k$ in a location $permit[n]$ ($permit$ is an array which is indexed by n). If a match attempt is initiated at some node n' (above n), then the match attempt will not continue (down in the tree) past node n unless $n' \leq permit[n]$. This can be done safely since all matches that began lower than $permit[n]$ cannot possibly lead to a match below n.

To implement this, we use the following mainline[2]:

```
for n : n ∈ nodes ∧ n.isleaf ⟶
    permit[n] := n.level
rof;
lev := (MAX n : n ∈ nodes : n.level);
do lev ≥ 0 ⟶
    for n : n ∈ nodes ∧ n.level = lev ⟶
        permit[n] := n.level − AM(q_0, n, lev)
    rof;
    lev := lev − 1
od
```

Correspondingly, we make use of the shift function *shift* which gives the shift distance in levels in the tree[3].

[2]To abort a match attempt when it is futile, we also pass a third argument to AM — the level at which the match attempt was started. The procedure now returns the integer shift (in terms of levels).

[3]The distance in levels is the ceiling of half of the distance given by the Commentz-Walter shift functions, since the shift distance given for the pattern path strings will be in terms of the path strings and a path string may be up to twice as long as the level of the leaf at which it ends because path strings contain the edge numbers interspersed with node labels.

proc $AM(q, n, beginlev)$ **returns** sh **is**
    $RM(q, n)$;
    if $\tau(q, n.label) = \perp \vee beginlev > permit[n]$ **then**
        $sh := shift(q)$
    else
        $q := \tau(q, n.label)$;
        $RM(q, n)$;
        if $n.arity = 0$ **then**
            $sh := shift(q)$
        else
            for $i \in [1, n.arity] \longrightarrow$
                if $\tau(q, i) = \perp$ **then**
                    $sh := sh$ **min** $shift(q)$
                else
                    $sh := sh$ **min** $AM(\tau(q, i), n.child(i), beginlev)$
                fi
            rof
        fi
    fi
corp

5 Conclusions

I have outlined a Watson-Watson type algorithm for regular tree pattern matching, thereby backing-up the statement accompanying my dissertation [Wats95]. Unfortunately, this algorithm has not yet been implemented and so nothing is known about its running time performance in practice, though it could potentially be proportional to the product of the input tree size and the size of the largest pattern tree. It therefore appears that the algorithm will not be as efficient as the KMP type algorithm (solving the same problem) given by Aho and Ganapathi in [1], which runs in time linear to the size of the input tree.

Like the other classes of BM type algorithms (for single keyword, multiple keyword, or regular expression string pattern matching), there are likely to be other (perhaps more efficient) variants of this algorithm. For example, it may be possible to devise such an algorithm which operates in a top-down manner, instead of in a bottom-up manner. Alternatively, it may be possible to reduce the primary problem in a different way than what we have done here. These alternatives are left as an exercise for the reader.

Acknowledgements:

I would like to thank Richard Watson (co-developer of the Watson-Watson regular expression pattern matching algorithm for strings) and Nanette Saes for their assistance in preparing this note.

References

[1] Aho, A.V. and M. Ganapathi. Efficient tree pattern matching: an aid to code generation, in: *Proceedings of the 12th ACM Symposium on Principles of Programming Languages,* New Orleans, p. 334–340, 1985.

[2] Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms,* Ph.D dissertation, Faculty of Computing Science, Eindhoven University of Technology, The Netherlands, 1995, ISBN 90-386-0396-7.

[3] Watson, B.W. A new regular grammar pattern matching algorithm, in: Diaz, J. and M. Serna, eds., *Proceedings of the European Symposium on Algorithms*, Barcelona, Spain, 1996.

[4] Watson, B.W. A collection of new regular grammar pattern matching algorithms, in: J. Holub, ed., *Proceedings of the First Annual Prague Stringology Club Workshop*, Prague, Czech Republic, 1996.

Simulation of *NFA* in Approximate String and Sequence Matching[1]

Jan Holub

Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo náměstí 13
121 35 Prague 2
Czech Republic

e-mail: holub@cs.felk.cvut.cz

Abstract. We present detailed description of simulation of nondeterministic finite automata (*NFA*) for approximate string matching. This simulation uses bit parallelism and used algorithm is called Shift-Or algorithm. Using knowledge of simulation of *NFA* by Shift-Or algorithm we design modification of Shift-Or algorithm for approximate string matching using generalized Levenshtein distance and modification for exact and approximate sequence matching.

Key words: finite automata, approximate string matching, simulation of nondeterministic finite automata, bitwise parallelism

1 Introduction

Approximate string matching is defined as a searching for all occurrences of pattern $P = p_1p_2 \dots p_m$ in text $T = t_1t_2 \dots t_n$ with at most k errors allowed. The number of errors allowed in a found substring is determined by a distance which is defined as a minimal number of edit operations needed to convert pattern P to the found substring. In the Hamming distance, the allowed edit operation is *replace* (replacing a character by another character). In the Levenshtein distance, the allowed edit operations are *replace*, *delete* (deletion of a character from the pattern) and *insert* (insertion of a character into the pattern). In the generalized Levenshtein distance, there is, besides edit operations *replace*, *delete* and *insert*, a new operation *transpose* (two adjacent characters are exchanged). This new edit operation represents situation when one types two characters in reversed order.

Sequence matching is defined as a searching for all occurrences of pattern $P = p_1p_2 \dots p_m$ in text $T = t_1t_2 \dots t_n$ such that between symbols p_i and p_{i+1}, $0 < i < m$, in text T can be located any number of input symbols. For approximate sequence matching we can also use Hamming, Levenshtein and generalized Levenshtein distance.

Nondeterministic finite automaton (*NFA*) is a quintuple (Q, A, δ, q_0, F), where Q is a set of states, A is a set of input symbols, δ is a mapping $Q \times (A \cup \{\varepsilon\}) \mapsto$ subsets of Q, q_0 is an initial state and F is a set of final states.

[1]This work was supported by grant FRVŠ 0892/97.

2 Exact String Matching

NFA for exact string matching is shown in Figure 1, where $m = 4$. Shift-Or algorithm [BG92], [WM92] for exact string matching uses one m-bit vector R in which l^{th} bit, $1 \leq l \leq m$ corresponds to l^{th} state of *NFA*. If l^{th} state is active then l^{th} bit is set to `0` and if l^{th} state is not active then l^{th} bit is set to `1`.

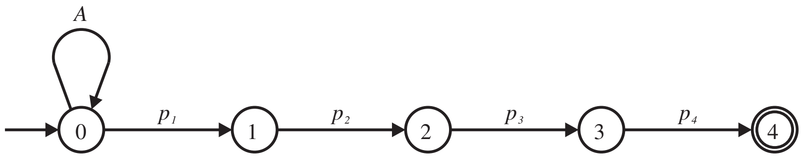


Figure 1: *NFA* for exact string matching.

In this *NFA* each nonfinal and noninitial state has transition to its right-hand neighbour. Hence transitions of all active states can be performed at once by bitwise operation `shift`. Now, it is necessary to select only those transitions that correspond to the input symbol t. It is handled by bitwise operation `or` with mask vector corresponding to the input symbol. These mask vectors are stored in mask table D. For each symbol a of input alphabet A there is one vector in which `0` is located in the same positions in which symbol a is located in the pattern P. The self-loop of the initial state is implemented by operation `shift` which inserts `0` at the beginning of the vector R. Before reading the first symbol of the input text the vector R is filled up by `1`. The formula for computing the vector when reading $(i+1)^{th}$ input symbol is (1)[2]. Shift-Or algorithm reports "pattern found" when m^{th} bit of the vector is set to `0`.

$$R_{i+1} \quad = \quad (shl(R_i) \, or D[t_{i+1}]) \tag{1}$$

Each version of Shift-Or algorithm described in this paper needs space $\mathcal{O}(\lceil \frac{m}{w} \rceil * \min(|A|, m+1))$ for mask table D, where $|A|$ denotes size of input alphabet A and w denotes length of computer word in bits. For exact string matching, space complexity of vector R is $\mathcal{O}(\lceil \frac{m}{w} \rceil)$ and time complexity is $\mathcal{O}(\lceil \frac{m}{w} \rceil * n)$, where n is a length of the input text. Moreover at the beginning of searching we can use faster trivial searching for the first character of the pattern and when it is found then we start Shift-Or algorithm.

3 Approximate String Matching

3.1 Hamming Distance

NFA for approximate string matching using Hamming distance was shown in [Me95]. Such *NFA* for $m = 4$ and number of errors allowed $k = 3$ is shown in Figure 2 where symbol $\overline{p_i}$ represents any symbol of input alphabet A except symbol p_i. Each

[2]In Shift-Or algorithm, transitions from states in our figures are implemented by operation *shl* (shift to the left) because of easier implementation in case that number of states of *NFA* is greater than length of computer word and vector R has to be divided into two vectors.

level of states represents number of errors allowed. Operation *replace* is represented by transition that leads to the following state of the level of one more errors. This transition has the same direction for all states so we can use also operation `shift` applied to previous value of vector for one lower number of errors.

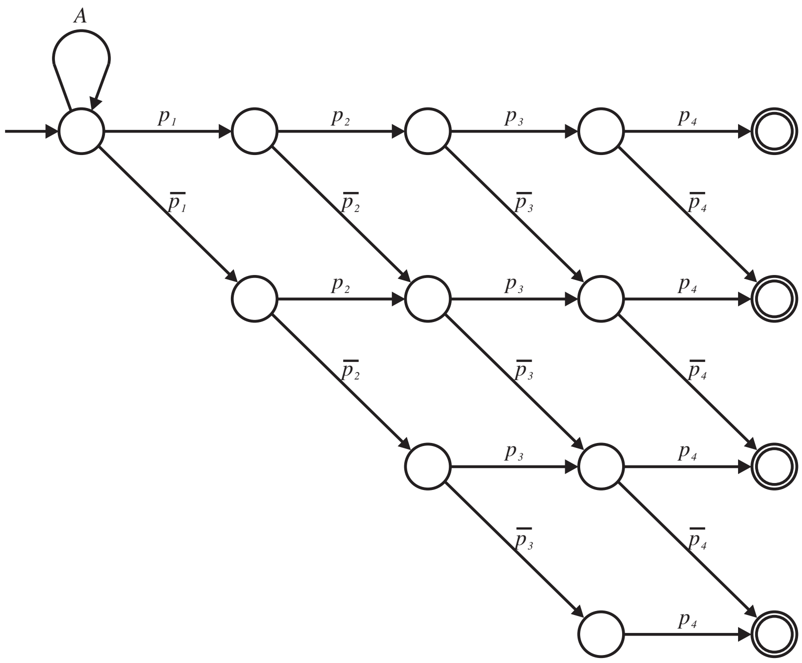


Figure 2: *NFA* for approximate string matching using Hamming distance.

Shift-Or algorithm uses for each level j, $0 \leq j \leq k$, of states one vector R^j. Vector R^0 is computed using formula (1). Vectors R^j, $j > 0$, allowing errors have to be computed with respect to transitions representing edit operation *replace*. They are computed using formula (2).

$$R^j_{i+1} \quad = \quad (shl(R^j_i)\, or D[t_{i+1}])\, and\, (shl(R^{j-1}_i) \qquad (2)$$

This formula does not correspond exactly to the *NFA* because transition representing edit operation *replace* is performed for both unmatching and matching symbols. Vector of states in previous level is only shifted but it should be also masked by negation of $D[t_{i+1}]$ in order to select only the transitions for unmatching symbols. Since we always search for minimal number of errors this simplification does not influence result.

In this case vectors R^j need space $\mathcal{O}(\lceil \frac{m}{w} \rceil * (k+1))$ and the algorithm runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil * n * (k+1))$.

3.2 Levenshtein Distance

NFA for approximate string matching using Levenshtein distance was shown in [Me96-1] and reduced in [Ho96]. Such *NFA* for $m = 4$ and number of errors allowed $k = 3$ is shown in Figure 3. There we can see edit operation *insert* which is represented by vertical transition — unmatching character inserted into the pattern. In Shift-Or algorithm, this transition is implemented by adding previous value of the vector for one lower number of errors. It is located at the end of formula (3) for computing the vector. Like for edit operation *replace*, this transition is also made for both unmatching and matching symbols but it also does not influence the result.

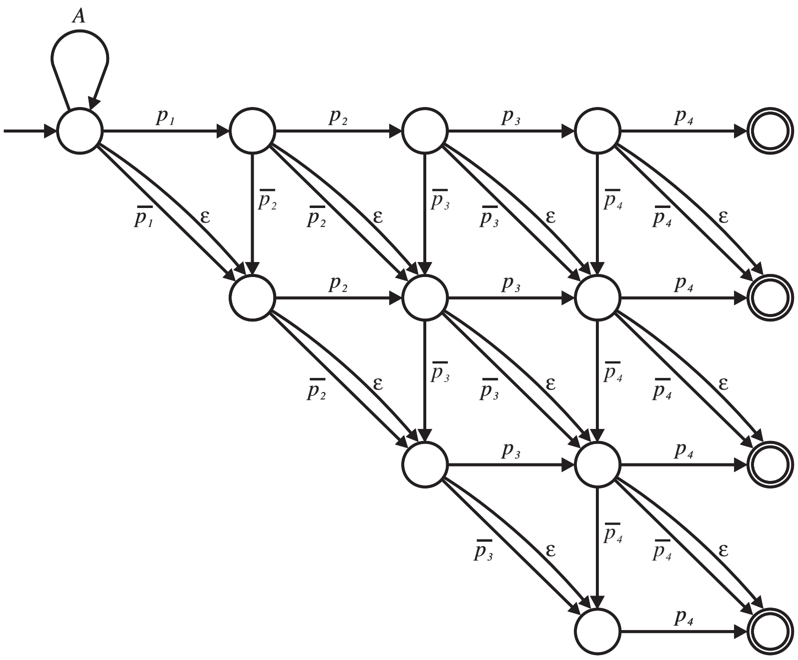


Figure 3: *NFA* for approximate string matching using Levenshtein distance.

$$R^j_{i+1} \;=\; (shl(R^j_i)\, or\, D[t_{i+1}])\, and\, (shl(R^{j-1}_i\, and\, R^{j-1}_{i+1}))\, and\, (R^{j-1}_i) \qquad (3)$$

In the *NFA* the edit operation *delete* is represented by ε-transition — any symbol deleted from the pattern. In Shift-Or algorithm, matching transitions are made for all active states and then resulting active states are moved to their right neighbours in the level for one higher number of errors. In formula (3) it is implemented by $shl(R^{j-1}_{i+1})$.

In this case vectors R^j need space $\mathcal{O}(\lceil\frac{m}{w}\rceil * (k+1))$ and the algorithm runs in time $\mathcal{O}(\lceil\frac{m}{w}\rceil * n * (k+1))$.

3.3 Generalized Levenshtein Distance

In approximate string matching using generalized Levenshtein distance we have new edit operation *transpose* that represents the situation when two adjacent symbols $p_i p_{i+1}$, $0 < i < m$, of the pattern P are placed in the found string in reversed order $p_{i+1} p_i$. In case of this edit operation we read two characters therefore this edit operation has to be represented by auxiliary state such that transition labeled by p_{i+1} leads to this state and transition labeled by p_i leads from this state. *NFA* for approximate string matching using generalized Levenshtein distance for $m = 4$ and $k = 3$ is shown in Fig. 4.

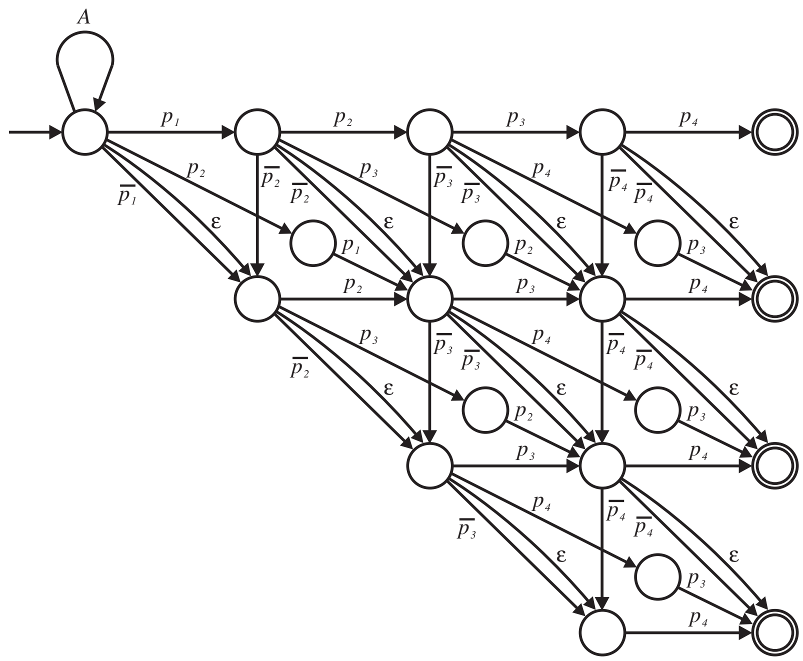


Figure 4: *NFA* for approximate string matching using generalized Levenshtein distance.

In Shift-Or algorithm we have to introduce new vectors S_j, $0 \leq j < k$, for auxiliary states. For each state q_i, whose right-hand neighbour q_{i+1} is not final state, holds that transition leading from this state to an auxiliary state q_{s_i} is labeled by matching symbol of state q_{i+1} and transition leading from the auxiliary state q_{s_i} is labeled by matching symbol of state q_i.

$$
\begin{aligned}
R_{i+1}^{j} &= (shl(R_i^j)\, or\, D[t_{i+1}])\, and\, (shl(R_i^{j-1}\, and\, R_{i+1}^{j-1}))\, and\, (R_i^{j-1}) \\
&\quad and\, (shl(S_i^{j-1}\, or\, D[t_{i+1}])) &\quad (4)\\
S_{i+1}^{j} &= shl(R_i^j)\, or (shr(D[t_{i+1}])) &\quad (5)
\end{aligned}
$$

Vectors R^j, $0 < j \leq k$, are computed by using formula (4) and vectors S^j, $0 \leq j < k$, are computed by using formula (5). At the beginning vectors S^j are filled up by `1`.

In this case vectors R^j and S^j both together need space $\mathcal{O}(\lceil \frac{m}{w} \rceil * (2k+1))$ and the algorithm runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil * n * (2k+1))$.

4 Sequence Matching

In previous sections we have shown simulation of *NFA*s for exact and approximate string matching. Since *NFA*s for exact and approximate sequence matching are very similar to corresponding *NFA*s for string matching we can use Shift-Or algorithm for sequence matching as well.

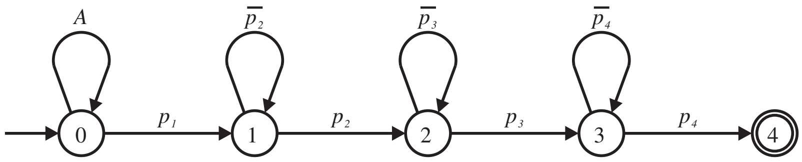


Figure 5: *NFA* for exact sequence matching.

NFA for sequence matching can be constructed from corresponding *NFA* for string matching by adding self-loop for mismatching symbols into all noninitial and nonfinal states. Such *NFA* for $m = 4$ is shown in Fig. 5. When constructing formula for computing vector R we use fromula for exact string matching (1) to which we have to add the part which represents self-loops for mismatching symbols. Self-loop expresses that a state is active even in the next step if mismatching symbol appears in the input. We implement it by adding previous value of vector R which is masked by negation of $D[t_{i+1}]$. The resulting formula is (6). This formula does not exactly correspond to *NFA* because it gives self-loop also into final state. Therefore when any final state reports "pattern found" the bit corresponding to this final state has to be set to `1`. It is faster than reseting the bit in formula (6) which is performed for each input symbol t_i.

$$R_{i+1} \quad = \quad (shl(R_i)\, or D[t_{i+1}])\, and\, (R_i \; or \; shr(not \; D[t_{i+1}])) \tag{6}$$

NFA for approximate sequence matching using Hamming distance is shown in Fig. 6 and formula for computing vector R is (7), for Levenshtein distance it is shown in Fig. 7 and formula is (9) and for generalized Levenshtein distance it is shown in Fig. 8 and formulae are (10) and (11).

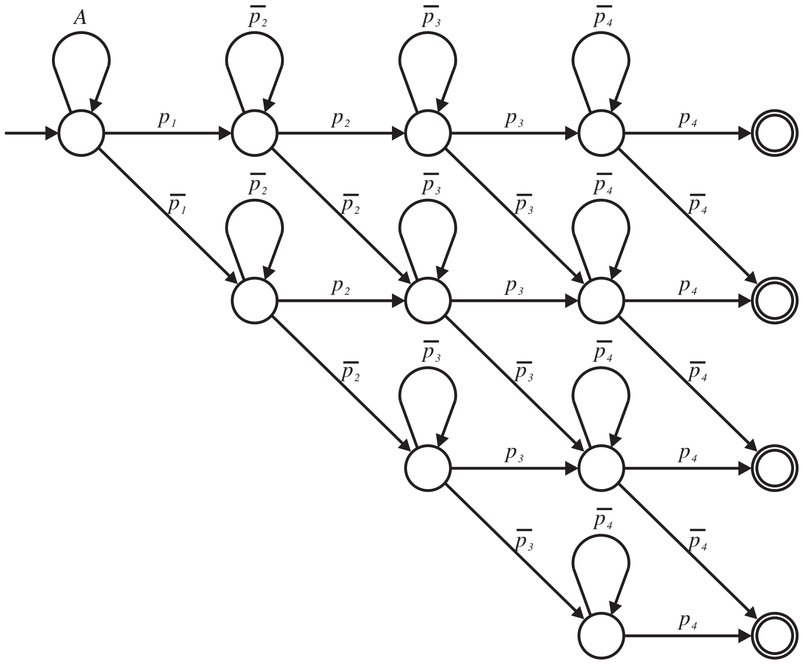


Figure 6: *NFA* for approximate sequence matching using Hamming distance.

$$
\begin{aligned}
R^j_{i+1} &= (shl(R^j_i)\, or\, D[t_{i+1}])\, and\, (shl(R^{j-1}_i))\, and\, (R_i \ or \ shr(not \ D[t_{i+1}])) && (7)\\
R^j_{i+1} &= (shl(R^j_i)\, or\, D[t_{i+1}])\, and\, (shl(R^{j-1}_i \ and\, R^{j-1}_{i+1}))\, and\, (R^{j-1}_i) && (8)\\
&\quad and\, (R_i \ or \ shr(not \ D[t_{i+1}])) && (9)\\
R^j_{i+1} &= (shl(R^j_i)\, or\, D[t_{i+1}])\, and\, (shl(R^{j-1}_i \ and\, R^{j-1}_{i+1}))\, and\, (R^{j-1}_i)\\
&\quad and\, (shl(S^{j-1}_i \ or\, D[t_{i+1}]))\, and\, (R_i \ or \ shr(not \ D[t_{i+1}])) && (10)\\
S^j_{i+1} &= shl(R^j_i)\, or (shr(D[t_{i+1}]))\, and\, (S_i \ or \ shr(shr(not \ D[t_{i+1}]))) && (11)
\end{aligned}
$$

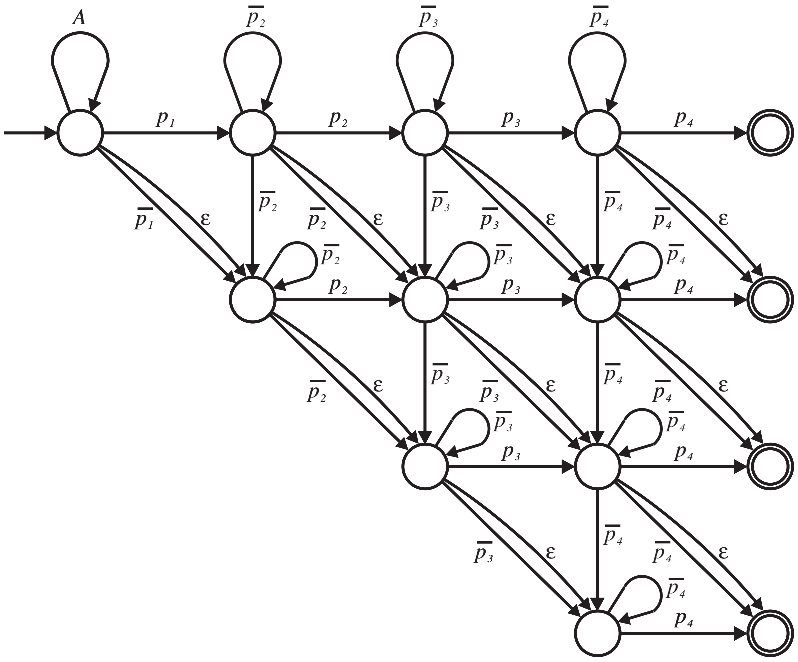


Figure 7: *NFA* for approximate sequence matching using Levenshtein distance.

Conclusions

We have presented detailed description of Shift-Or algorithm that runs in time $\mathcal{O}(\lceil \frac{m}{w} \rceil * n * k)$ and needs space $\mathcal{O}(\lceil \frac{m}{w} \rceil * (2k+1))$. Besides simulation of *NFA*s we can transform them to corresponding deterministic finite automata (*DFA*s) that run in time $\mathcal{O}(n)$. In case of exact string matching the number of states of *DFA* for pattern P is the same as the number of states of *NFA* for pattern P so except for very short text it is faster to use *DFA*.

In case of approximate string and sequence matching the number of states of *DFA* unfortunately seems to be exponential but exact bounds have not been determined. Some estimations were presented in [Me96-2] but they also seem to be pessimistic. In case that we do not want to know the number of errors in the found string we can reduce *NFA*, shorten vectors R^j and simplify formulae as shown in [Ho96].

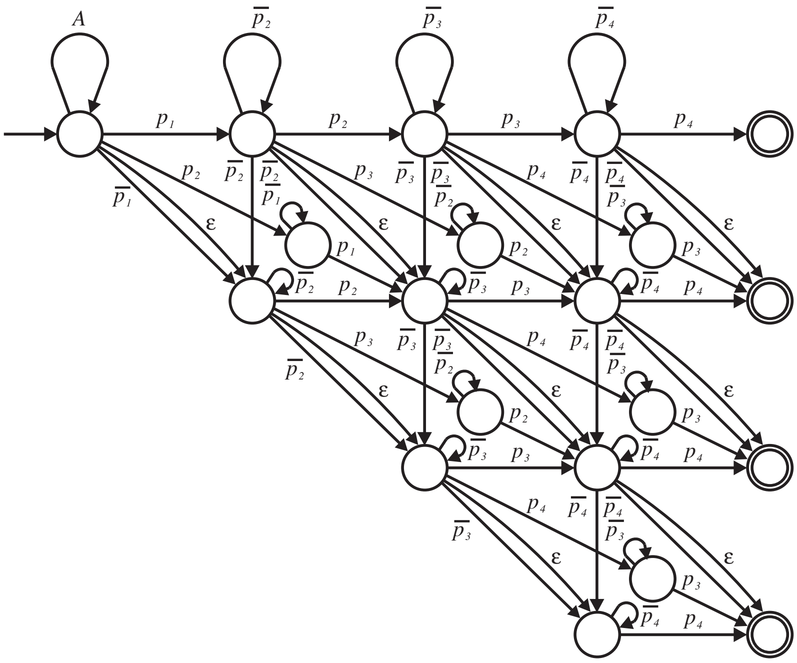


Figure 8: *NFA* for approximate sequence matching using generalized Levenshtein distance.

When searching it is necessary to consider length of the input text, length of the pattern and memory space available in order to choose optimal searching method — *DFA* or simulation of *NFA*.

References

[BG92] Baeza-Yates, R., Gonnet, G.H.: A New Approach to Text Searching. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 74–82.

[Ho96] Holub, J.: Reduced Nondeterministic Finite Automata for Approximate String Matching. Proceedings of the Prague Stringology Club Workshop '96, Czech Technical University, August 1996, pp. 19–27.

[Me95] Melichar, B.: Approximate String Matching by Finite Automata. Computer Analysis of Images and Patterns, LNCS 970, Springer-Verlag, Berlin 1995, pp. 342–349.

[Me96-1] Melichar, B.: String Matching with k Differences by Finite Automata. Proceedings of the 13^{th} ICPR, Vol. II, August 1996, pp. 256–260.

[Me96-2] Melichar, B.: Space Complexity of Linear Time Approximate String Matching. Proceedings of the Prague Stringology Club Workshop '96, Czech Technical University, August 1996, pp. 28–36.

[WM92] Wu, S., Manber, U.: Fast Text Searching Allowing Errors. Communications of the ACM, October 1992, Vol. 35, No. 10, pp. 83–91.

SPARE Parts: A C++ Toolkit for String PAttern REcognition

Bruce W. Watson

Ribbit Software Systems Inc.
(IST Technologies Research Group)
Box 24040, 297 Bernard Ave.
Kelowna, B.C., V1Y 9P9, Canada

e-mail: watson@RibbitSoft.com

Abstract. In this paper, we consider the design and implementation of SPARE Parts, a C++ toolkit for pattern matching. SPARE Parts is the second generation string pattern matching toolkit from the Ribbit Software Systems Inc. and the Eindhoven University of Technology. The toolkit contains implementations of the well-known Knuth-Morris-Pratt, Boyer-Moore, Aho-Corasick and Commentz-Walter algorithms (and their variants).

The toolkit is publicly available (though it is not in the public domain and it may not be redistributed) for noncommercial use. A totally re-implemented toolkit, known as SPARE Parts II, is available for commercial licensing from Ribbit Software Systems Inc. In addition to the functionality of SPARE Parts, it contains approximate pattern matchers, regular pattern matchers, multi-dimensional pattern matchers, and a highly tuned set of foundation classes.

Key words: keyword pattern matching, C++ toolkits, C++ frameworks, generic algorithms, foundation classes, taxonomies.

1 Introduction and related work

In this paper, we outline the design, design rationale, and use of SPARE Parts, a C++ toolkit for string pattern matching. The algorithms implemented in the toolkit include:

- The Knuth-Morris-Pratt [KMP77] single keyword pattern matching algorithm.
- Several variants of the Boyer-Moore [BM77] single keyword pattern matching algorithms.
- Several variants of the Aho-Corasick [AC75] multiple keyword pattern matching algorithms.
- Several variants of the Commentz-Walter [Com79a, Com79b] multiple keyword pattern matching algorithm.

In addition to the original papers, all of these algorithms are extensively treated (with full correctness arguments) in a taxonomy presented in [Wats95, Chapter 4].

SPARE Parts is the second generation string pattern matching toolkit from the Ribbit Software Systems Inc. and the Eindhoven University of Technology. The toolkit is publicly available (though not in the public domain and it may not be redistributed) for noncommercial use. A re-implemented (and greatly extended) toolkit, known as SPARE Parts II, is available for commercial licensing from Ribbit Software Systems Inc. In addition to the functionality of SPARE Parts, it contains approximate pattern matchers, regular pattern matchers, multi-dimensional pattern matchers, and a highly tuned set of foundation classes.

The first generation toolkit (called the Eindhoven Pattern Kit, written in C and described in [Wats94, Appendix A]) is a procedural library based upon the original taxonomy of pattern matching algorithms [WZ92]. Experience with the Eindhoven Pattern Kit revealed a couple of deficiencies (leading to the design of SPARE Parts), detailed as follows. The rudimentary and explicit memory management facilities in C caused numerous errors in the code and made it difficult to perform pattern matching over more than one string simultaneously (in separate threads of the program) without completely duplicating the code. While the performance of the toolkit was excellent, some of the speed was due to sacrifices made in the understandability of the client interface.

There are other existing pattern matching toolkits, notably the toolkit of Hume and Sunday [HS91]. Their toolkit consists of a number of implementations of Boyer-Moore type algorithms — organized so as to form a taxonomy of the Boyer-Moore family of algorithms. Their toolkit was primarily designed to collect performance data on the algorithms. As a result, the algorithms are implemented (in C) for speed and they sacrifice some of the safety (in terms of error checking) that would normally be expected of a general toolkit. Furthermore, the toolkit does not include any of the non-Boyer-Moore pattern matching algorithms (other than a brute-force pattern matcher) and, most noticeably, does not contain multiple keyword pattern matchers.

SPARE Parts is a completely object-oriented implementation of the algorithms appearing in [Wats95, Chapter 4]. SPARE Parts is designed to address the shortcomings of both of the toolkits described above. The following are the primary features of the class library:

- The design of SPARE Parts follows the structure of the taxonomy in [Wats95, Chapter 4] very closely. As a result, the code is easier to understand and debug. In addition, SPARE Parts includes implementations of almost all of the algorithms described in [Wats95, Chapter 4].

- The use of C++ (instead of C) for the implementation has helped to avoid many of the memory management-related bugs that were present in the original toolkit.

- The client interface to the toolkit is particularly easy to understand and use. The flexibility introduced into the interface does not reduce the performance of the code in any significant way.

- The toolkit supports multi-threaded use of a single pattern matching object.

This paper is structured as follows:

- Section 2 considers issues in the design and implementation of class libraries.
- Section 3 gives an introduction to the client interface of the toolkit. It includes some examples of programs which use SPARE Parts.
- Section 4 presents some experiences with the toolkit and the conclusions of this chapter.
- Section 5 gives some information on how to obtain and compile the toolkit.

2 Designing and implementing class libraries

In this section, we briefly discuss some of the issues involved in designing, implementing, and presenting class libraries (or *toolkits*). The following description of a toolkit is taken from [GHJV95, p. 26]:

> A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job. They are the object-oriented equivalent of subroutine libraries.

We will use the terms *class library, library,* and *toolkit* interchangeably. We will also use the term *client* to refer to a program that makes use of classes in the toolkit, or the author of such a program. The important aspects and design goals of a toolkit are:

- Toolkits do not provide a user interface. (Toolkits that do provide user interfaces should be placed in the category of 'application program'.)
- The classes in the toolkit must have a coherent design, meaning that they are designed and coded in the same style. They have a clear relationship and a logical class hierarchy.
- The client interface to the library must be easily understood, permitting clients to make use of the library with a minimum of reading.
- The efficiency of using the classes in the toolkit must be comparable to hand-coded special-purpose routines — the toolkit must be applicable to production quality software.
- To provide an educational use for the toolkits, and to allow clients to easily modify classes and member functions, the method of implementation must be clear and understandable.

The toolkit is implemented in the C++ programming language, which was chosen because of its object-oriented features and its widespread availability. Efforts were made to refrain from using obscure features of C++ (such as RTTI or name spaces), or language features not easily found in other object-oriented programming languages (such as multiple-inheritance).

Throughout this paper, we assume that the reader is familiar with the C++ language and object-oriented terminology (especially the C++ variety).

The general process of library design will not be described here, as there is a large body of literature discussing this issue. The following books are of particular relevance:

- [GHJV95, Souk94] discuss 'design patterns' (not related to our pattern matching problem) which are used heavily in library design.
- [CE95], [Stro91, Chapter 13] and [Stro94, Chapter 8] provide a general discussion of C++ library design.
- [MeyB94] is an excellent treatment of the design of a number of loosely coupled libraries in the Eiffel programming language. Many of the concepts and techniques discussed in the book are broadly applicable to C++ as well.
- [Plau95, Teal93] discuss the design and implementation of specific C++ libraries — the standard C++ library[1] and the IOStreams (input and output) class libraries respectively.

We define the following types of object-oriented classes:

User: A class intended for use by a client program.

Client: A class defined in the client program.

Implementation: A class defined in the toolkit for exclusive use by the toolkit. The class is used to support the implementation of the client classes.

Foundation: Those implementation classes which are simple enough to be reused in other (perhaps unrelated) class libraries.

Interface: An abstract (pure virtual) class which is declared to force a particular public interface upon its inheritance descendants.

Base: An inheritance ancestor of a particular class.

Derived: An inheritance descendant of a particular class.

2.1 Motivations for writing class libraries

There are a number of motivations for creating the class libraries:

- Until now, few general purpose toolkits of pattern matchers existed. The ones that do exist are not intended for general use in production quality software.

[1] Plauger's book considers the implementation of an early, and now defunct, draft of the standard library.

- The level of coherence normally required to implement a toolkit was not previously possible. The literature on pattern matching algorithms was scattered and in some places incomplete. With the construction of the taxonomy, [Wats95, Chapter 4], all of the algorithms are described in a coherent fashion, allowing us to base the class library structures on the taxonomy structure.
- The uniformity of implementation that was possible (given the taxonomy) had two important effects:
  - Clients need not examine the source code in order to make a decision on which class to use; the quality of the implementations of each of the pattern matchers is roughly the same.
  - Uniformity gives greater confidence in the accuracy of relative performance comparing different algorithms.
- The toolkit and the taxonomy can serve as examples of implementation techniques for class libraries; in particular methods for organizing template classes[2] and class hierarchies.
- Implementing the abstract algorithm can be painless and fun, given the taxonomy presentation of the algorithms and their correctness arguments.

2.2 Code sharing

One of the main aims of object-oriented programming is that it permits, and even encourages, code sharing (or code reuse). The code reuse in object-oriented programming corresponds neatly with the factoring of common parts of algorithms in the taxonomy.

Although code sharing can be achieved in a number of ways, in this section we discuss four techniques which could have been used in the design of the toolkits. The first discussion centres around the use of base classes (with virtual member functions) versus templates. The second discussion concerns the use of composition versus protected inheritance.

2.2.1 Base classes versus templates

A number of the pattern matching objects have common functionality and it seems wasteful to duplicate the code in each of the specific types of pattern matchers.

The obvious design involves creating a new base class and factoring the common code into the base class. Each of the pattern objects would then inherit from this base and provide specific virtual member functions to obtain the desired functionality. For example, the Commentz-Walter algorithms all share a common algorithm skeleton: they each have specific shift functions. We could create a CW base class with the functionality of the skeleton and provide a virtual 'shift distance' member function to obtain the Commentz-Walter variants.

[2]We use the term *template class*, as opposed to *class template* suggested by Carroll and Ellis in [CE95]. Our choice was made to correspond to the term *generic class* used in some other object-oriented programming languages.

The advantage of this approach is its elegance. It provides a relatively easy to understand class hierarchy, that reflects the structure of the taxonomy. Furthermore, a member function which takes (as a parameter) a pointer to a CW object need not know which particular variant (of a CW object) the pointer points to, only that the CW object satisfies the general CW functionality. This solution provides code reuse at both the source language and executable image levels. The disadvantage is that it would require a virtual function call for every shift. Indeed, if the same technique was used to factor the common code from the Aho-Corasick variants, it would require a virtual function call for every character of the input string.

The other approach is to create a template class CW, which takes a 'shifter class' as its template (type) parameter. We would then provide a number of such shifter classes, for use as template parameters — each giving rise to one of the Commentz-Walter variants. The primary advantage of this approach is that it is efficient: when used to implement the Aho-Corasick algorithms, each character in the input string will require a non-virtual function call (which may be inlined, unlike virtual function calls). The disadvantages are twofold: pointers to the variants of the CW algorithms are not interchangeable and code will be generated for each of the CW variants. The code reuse is at the source level and not at the executable image level.

It is expected, for example, that few clients of the toolkits will instantiate objects of different CW classes. A programmer writing an application using pattern matching is more likely to choose a particular type of pattern matcher, as opposed to creating objects of various different types. For this reason, the advantages of the template approach are deemed to outweigh its disadvantages, and we prefer to use it over base classes in the toolkits.

2.2.2 Composition versus protected inheritance

Composition (sometimes called the *has-a* relationship) and protected inheritance (sometimes called the *is-a* relationship) are two additional solutions to code sharing. We illustrate the differences between these two solutions using an example. When implementing a *Set* class, we may wish to make use of an already-existing *Array* class. There are two ways to do this: composition and protected inheritance.

With protected inheritance, class *Set* inherits from *Array* in a protected way. Class *Set* still gets the required functionality from *Array*, but the protected inheritance prevents the *is-a* relation between *Set* and *Array* (that is, we cannot treat a *Set* as an *Array*). The advantage of this approach is that it is elegant and it is usually the approach taken in languages such as Smalltalk and Objective-C [Budd91]. The disadvantage is that the syntax of C++ places the inheritance clause at the beginning of the class declaration of *Set*, making it plain to all clients of *Set* that it is implemented in terms of *Array*. Furthermore, protected inheritance (and indeed private inheritance) is one of the rarely-used corners of C++, and it is unlikely that the average programmer is familiar with it [MeyS92, Murr93].

In a composition approach, an object of class *Set* has (in its private section) an object of class *Array*. The *Set* member functions invoke the appropriate member functions of *Array* to provide the desired functionality. The advantage of this approach is that it places all implementation details in the private section of the class definition. The disadvantage is that it deviates from the accepted practice (in some other languages) of inheriting for implementation. It is, however, the standard approach in C++. At first glance, it would appear that composition can lead to some inefficiency: in our example, an invocation of a *Set* member function would, in turn, call an *Array* member function. These extra function calls, usually called *pass-throughs*, are frequently eliminated through inlining.

There are no efficiency-based reasons to choose one approach over the other. For this reason, we arbitrarily choose composition because of the potential readability and understandability problems with protected inheritance.

2.3 Coding conventions and performance issues

At this time, coding in C++ presents at least two problems: the language is not yet stable (it is still being standardized) and, correspondingly, the "standard" class libraries are not yet stable.

In designing the libraries, every effort was made to use only those language features which are well-understood, implemented by most compilers and almost certain to remain in the final language. Likewise, the use of classes from the proposed standard library, or from the Standard Template Library [SL94], was greatly restricted. A number of relatively simple classes (such as those supporting strings, arrays, and sets) were defined from scratch, in order to be free of library changes made by the standardizing committee. A future version of the toolkits will make use of the standard libraries once the International Standards Organization has approved the C++ standard.

In the object-oriented design process, it is possible to go overboard in defining classes for even the smallest of objects — such as alphabet symbols. In the interests of efficiency, we draw the line at this level and make use of integers for such basic objects.

Almost all of the classes in the toolkits have a corresponding class invariant member function, which returns *TRUE* if the class is structurally correct and *FALSE* otherwise[3]. Structural invariants have proven to be particularly useful in debugging and in understanding the code (the structural invariant is frequently a good first place to look when trying to understand the code of a class). For this reason, they have been left in the released code (they can be disabled as described in the next section).

We use a slightly non-traditional way of splitting the source code into files. The public portion of a class declaration is given in a `.hpp` file, while the private parts are included from a `.ppp` file. There is a corresponding `.cpp` file containing all of the out-of-line member function definitions. A `.ipp` file contains member functions which can be inlined for performance reasons. By default the member functions in the `.ipp` file are out-of-line. The inlining can be enabled by defining the macro `INLINING`. To implement such conditional inlining, the `.ipp` file is conditionally included into the `.hpp` or the `.cpp` file. The inlining should be disabled during debugging or for smaller executable images.

3 Using the toolkit

In this section, we describe the client interface of the toolkit and present some examples of programs using the toolkit.

[3] This will be changed to use the new **bool** datatype once most compiler support it.

The client interface defines two types of abstract pattern matchers: one for single keyword pattern matching and one for multiple keyword pattern matching. (A future version of SPARE Parts can be expected to include classes for regular expression pattern matching — for example, an implementation of the algorithm described in [Wats95, Chapter 5].) All of the single keyword pattern matching classes have constructors which take a keyword. Likewise, the multiple keyword pattern matchers have constructors which take a set of keywords. Both types of pattern matchers make use of *call-backs* (to be explained shortly) to register matched patterns. In order to match patterns using the call-back mechanism, the client takes the following steps (using single keyword pattern matching as an example):

1. A pattern matching object is constructed (using the pattern as the argument to the constructor).
2. The client calls the pattern matching member function *PMSingle::match*, passing the input string and a pointer *f* to a client defined function which takes an **int** and returns an **int**[4]. (This function is called the *call-back function.*)
3. As each match is discovered by the member function, the call-back function is called; the argument to the call is the index (into the input string) of the symbol immediately to the right of the match. (If there is no symbol immediately to the right, the length of the input string is used.)
4. If the client wishes to continue pattern matching, the call-back function returns the constant *TRUE*, otherwise *FALSE*.
5. When no more matches are found, or the call-back function returns *FALSE*, the member function *PMSingle::match* returns the index of the symbol immediately to the right of the last symbol processed.

We now consider an example of single keyword pattern matching.

The following program searches an input string for the keyword `hisher`, printing the locations of all matches along with the set of matched keywords:

```
#include "com-misc.hpp"
#include "pm-kmp.hpp"
#include <iostream.h>

static int report( int index ) {
    cout << index << '\n';
    return( TRUE );
}

int main( void ) {
    auto PMKMP Machine( "hisher" );
    Machine.match( "hishershey", &report );
    return( 0 );
}
```

[4]The integer return value is a Boolean value; recall that *TRUE* and *FALSE* have type **int** in C and C++. The new **bool** keyword is not yet supported by all compilers.

The header file `com-misc.hpp` provides a definition of constants *TRUE* and *FALSE*. Header file `pm-kmp.hpp` defines the Knuth-Morris-Pratt pattern matching class, while header file `iostream.h` defines the input and output streams, including the standard output *cout*. Function *report* is our call-back function, simply printing the index of the match (to the standard output) and returning *TRUE* to continue matching. The *main* function (the program mainline) creates a local KMP machine, with keyword `hisher`. The machine is then used to find all matches in string `hishershey`. (Recall that, in C and C++, a pointer to the beginning of the string is passed to member *match*, as opposed to the entire string.)

In addition to the KMP algorithm defined in `pm-kmp.hpp`, other single keyword pattern matchers are defined in header file `bms.hpp`, which contains suggestions for instantiating some of the Boyer-Moore variants. Additionally, a brute-force single keyword pattern matcher is defined in `pm-bfsin.hpp`.

Multiple keyword pattern matching is performed in a similar manner, as the following example shows. The following program searches an input string for the keywords `his`, `her`, and `she`, printing the locations of all matches:

```
#include "com-misc.hpp"
#include "string.hpp"
#include "set.hpp"
#include "acs.hpp"
#include <iostream.h>

static int report( int index, const Set<String>& M ) {
    cout << index << M << '\n';
    return( TRUE );
}

int main( void ) {
    auto Set<String> P( "his" );
    P.add( "her" ); P.add( "she" );
    auto PMACOpt Machine( P );
    Machine.match( "hishershey", &report );
    return( 0 );
}
```

Header file `string.hpp` defines a string class, while `set.hpp` defines a template class for sets of objects. Header file `acs.hpp` defines the Aho-Corasick pattern matching classes. Function *report* is our call-back function, simply printing the index of the match (to the standard output) and the set of keywords matching, and returning *TRUE* to continue matching. Note that the call-back function has a different signature for multiple keyword pattern matching: it takes the index of the symbol to the right of the match and the set of keywords matching with *index* as their right end-point.

The *main* function (the program mainline) creates a local AC machine from the keyword set. The machine is then used to find all matches in string `hishershey`. In the following two sections, we consider ways to use SPARE Parts more efficiently in certain application domains.

3.1 Multi-threaded pattern matching

One important design feature (as a result of the call-back client interface) of SPARE Parts is that it supports multi-threading. This can lead to high performance in applications hosted on multi-threading operating systems. For example, consider an implementation of a keyword `grep` application in which 1000 files are to be searched for occurrences of a given keyword. The following are three potential solutions:

- In a sequential solution, a single pattern matching object is constructed and each of the 1000 files are scanned (in turn) for matches.
- In a naïve multi-threaded solution, 1000 threads are created (each corresponding to one of the input files). Each of the threads construct a pattern matching object, which is then used to search the file.
- An efficient solution is to create a single matching object, with 1000 threads sharing the single object. Each of the threads proceeds to search its file, using its own invocation of member function *PMSingle::match*.

The last (most efficient) solution would not have been possible without the call-back client interface.

3.2 Alternative alphabets

The default structure in SPARE Parts is to make use of the entire ASCII character set as the alphabet. This can be particularly inefficient and wasteful in cases where only a subset of these letters are used. For example, in genetic sequence searching, only the letters a, c, g, and t are used. SPARE Parts facilitates the use of smaller alphabets through the use of *normalization*. Header file `alphabet.hpp` defines a constant *ALPHABETSIZE* (which, by default is *CHAR_MAX*). The alphabet which SPARE Parts uses is the range $[0, ALPHABETSIZE)$. An alternative alphabet can be used by redefining *ALPHABETSIZE* and mapping the alternative alphabet in the required range. The mapping is performed by functions *alphabetNormalize* and *alphabetDenormalize*, both declared in `alphabet.hpp` (by default, these functions are the identity functions). The only requirement is that the functions map 0 to 0 (this is used to identify the end of strings). In the genetic sequence example, we would make use of the following version of header `alphabet.hpp`:

```
#include <assert.h>
#define ALPHABETSIZE 5

inline char alphabetNormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 'a':   return( 1 );
    case 'c':   return( 2 );
    case 'g':   return( 3 );
    case 't':   return( 4 );
    default:    assert( !"Non-genetic character" );
    }
```

```
}

inline char alphabetDenormalize( const char a ) {
    switch( a ) {
    case 0:     return( 0 );
    case 1:     return( 'a' );
    case 2:     return( 'c' );
    case 3:     return( 'g' );
    case 4:     return( 't' );
    default:    assert( !"Non-genetic image" );
    }
}
```

4 Experiences and conclusions

Designing and coding SPARE Parts lead to a number of interesting experiences in class library design. In particular:

- SPARE Parts comprises 5787 lines of code in 59 `.hpp`, 32 `.cpp`, 43 `.ppp`, and 49 `.ipp` files.
- Compiling the files, with the WATCOM C++32 Version 9.5b compiler, shows that the size of the object code varies very little for the various types of pattern matchers.
- The taxonomy presented in [Wats95, Chapter 4] was critical to correctly implementing the many complex precomputation algorithms.
- Designing and structuring generic software (reusable software such as class libraries) is much more difficult than designing software for a single application. The general structure of the taxonomy proved to be helpful in guiding the structure of SPARE Parts.
- In [Wats95, Chapter 4], we consider the relative performance of the algorithms implemented in SPARE Parts. It is also helpful to consider how the implementations in SPARE Parts fare against commercially available tools such as the `fgrep` program. Four `fgrep`-type programs were implemented (using SPARE Parts), corresponding to the Knuth-Morris-Pratt, Aho-Corasick, Boyer-Moore and Commentz-Walter algorithms. The four tools were benchmarked informally against the `fgrep` implementation which is sold as part of the MKS toolkit for MS-DOS. The resulting times (to process a 984149 byte text file, searching for a single keyword) are:

`fgrep` *variant*	MKS	KMP	BM	AC	CW
Time (sec)	3.9	5.1	4.2	4.7	4.0

  These results indicate that using a general toolkit such as SPARE Parts will result in performance which is similar to carefully tuned C code (such as MKS `fgrep`). Much more extensive test results are reported in [Wats95, Wats96].

Detailed records were kept on the time required for designing, typing, compiling (and fixing syntax errors) and debugging the toolkit. The time required to implement the toolkit is broken down as follows (an explanation of each of the tasks is given below):

Task	Design	Typing	Compile/Syntax	Debug	Total
Time (hrs:min)	6:00	13:40	10:05	5:15	35:00

Most of these times are quite short compared to what a software engineer could expect to spend on a project of comparable size. The following paragraphs explain exactly what each of the tasks entailed:

- The *design* phase involved the creation of the inheritance hierarchy and the declaration (on paper) of all of the classes in the toolkit. (A C++ declaration provides names and signatures of functions, types and variables, whereas a definition provides the implementation of these items.) The design phase proceeded exceptionally smoothly thanks to a number of things:
  - The inheritance hierarchy followed directly from the structure of the taxonomy.
  - The decisions on the use of templates (instead of virtual functions) and call-backs were made on the basis of experience gained with FIRE Engine, Ribbit's finite automata and transducer toolkit. These decisions were also somewhat forced by the efficiency requirements for the toolkit as well as the need for multi-threading.
  - Representation issues, such as the selection of data structures, were resolved using experience gained with the earlier Eindhoven Pattern Kit.
- Once the foundation classes were declared and defined, typing the code amounted to a simple translation of guarded commands (appearing in [Wats95]) to C++.
- The times required for compilation and syntax checking were minimized by using a very fast integrated environment (BORLAND C++) for initial development. Only the final few compilations were done using the (slower, but more thoroughly optimizing) WATCOM C++ compiler. The advantages of using a fast development environment on a single user personal computer should not be underestimated.
- Since the C++ code in the toolkit was implemented directly from the abstract algorithms (for which correctness arguments are given), the only (detected) bugs were those involving typing errors (such as the use of the wrong variable, etc.). Correspondingly, little time needed to be spent on debugging the toolkit.

New research in pattern matching requires that tools such as SPARE Parts evolve. The following are some of the upcoming changes:

- The toolkit will use a version of the C++ Standard Template Library.
- In light of the success and widespread applicability of the commercialized version, SPARE Parts II, SPARE Parts will be template parameterized to support input strings and patterns over arbitrary alphabets (as opposed to the rudimentary alphabet support now provided).

- Efforts will begin to integrate approximate pattern matching algorithms into the toolkit.

5 Obtaining and compiling the toolkit

SPARE Parts is available via `www.RibbitSoft.com/research/watson/`. The toolkit and some associated documentation are combined into a `tar` file.

SPARE Parts has been successfully compiled with BORLAND C++ Versions 3.1 and 4.0, and WATCOM C++32 Version 9.5b on MS-DOS and MICROSOFT WINDOWS 95 platforms. Since the WATCOM compiler is also a cross-compiler, there is every reason to believe that the code will compile for WINDOWS NT or for IBM OS/2. The implementation of the toolkit makes use of only the most basic features of C++ and it should be compilable using any of the template-supporting UNIX based C++ compilers.

A version of SPARE Parts will remain freely available (though not in the public domain). Contributions to the toolkit, in the form of new algorithms or alternative implementations, are welcome.

References

[AC75] AHO, A.V. and M.J. CORASICK. Efficient string matching: an aid to bibliographic search, *Comm. ACM,* **18**(6) (1975) 333–340.

[BM77] BOYER, R.S. and J.S. MOORE. A fast string searching algorithm, *Comm. ACM,* **20**(10) (1977) 62–72.

[Budd91] BUDD, T.A. *An introduction to object-oriented programming.* (Addison-Wesley, Reading, MA, 1991).

[CE95] CARROLL, M.D. and M.A. ELLIS. *Designing and coding reusable C++.* (Addison-Wesley, Reading, MA, 1995).

[Com79a] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, in: H.A. Maurer, ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer-Verlag, Berlin, 1979) 118–132.

[Com79b] COMMENTZ-WALTER, B. A string matching algorithm fast on the average, Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.

[GHJV95] GAMMA, E., R. HELM, R. JOHNSON, and J. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, Reading, MA, 1995).

[HS91] HUME, S.C. and D. SUNDAY. Fast string searching, *Software—Practice and Experience* **21**(11) (1991) 1221–1248.

[KMP77] KNUTH, D.E., J.H. MORRIS and V.R. PRATT. Fast pattern matching in strings, *SIAM J. Comput.* **6**(2) (1977) 323–350.

[MeyB94] MEYER, B. *Reusable Software: The Base Object-Oriented Component Libraries.* (Prentice Hall, Englewood Cliffs, NJ, 1994).

[MeyS92] MEYERS, S. *Effective C++: 50 specific ways to improve your programs.* (Addison-Wesley, Reading, MA, 1992).

[Murr93] MURRAY, R.B. *C++ strategies and tactics.* (Addison-Wesley, Reading, MA, 1993).

[Plau95] PLAUGER, P.J. *The Draft Standard C++ Library.* (Prentice Hall, New Jersey, 1995).

[SL94] STEPANOV, A. and M. LEE. Standard Template Library, Computer Science Report, Hewlett-Packard Laboratories, 1994.

[Souk94] SOUKUP, J. *Taming C++: Pattern Classes and Persistence for Large Projects.* (Addison-Wesley, Reading, MA, 1994).

[Stro91] STROUSTRUP, B. *The C++ programming language.* (Addison-Wesley, Reading, MA, 2nd edition, 1991).

[Stro94] STROUSTRUP, B. *The Design and Evolution of C++.* (Addison-Wesley, Reading, MA, 1994).

[Teal93] TEALE, S. *C++ IOStreams Handbook.* (Addison-Wesley, Reading, MA, 1993).

[Wats94] WATSON, B.W. The performance of single-keyword and multiple-keyword pattern matching algorithms, Computing Science Report 94/19, Eindhoven University of Technology, The Netherlands, 1994.

[Wats95] WATSON, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* (Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995).
For availability, see `www.RibbitSoft.com/research/watson/`.

[Wats96] WATSON, B.W. The performance of single and multiple keyword pattern matching algorithms, *Workshop on String Processing* (Recife, Brazil, 1996). Available via `www.RibbitSoft.com/research/watson/`.

[WZ92] WATSON, B.W. and G. ZWAAN. A taxonomy of keyword pattern matching algorithms, Computing Science Report 92/27, Eindhoven University of Technology, The Netherlands, 1992.

[WZ95] WATSON, B.W. and G. ZWAAN. A taxonomy of sublinear keyword pattern matching algorithms, Computing Science Report 95/13, Eindhoven University of Technology, The Netherlands, 1995.

Algebra of Pattern Matching Automata

Václav Snášel, Tomáš Koutný

Department of Computer Science
Palacky University
Tomkova 40
771 00 Olomouc
Czech Republic

e-mail: {Vaclav.Snasel, Tomas.Koutny}@upol.cz

Abstract. In this paper we classify pattern matching problems using algebraic means. We construct an algebra of pattern matching automata in which finite automata are the elements and operations applied to them correspond to the creation of new pattern matching problems. We present several such operations and describe some identified properties of the algebra defined in this way.

Key words: pattern matching, finite automata, algebra

1 Introduction

In this paper we will classify pattern matching problems using algebraic means. We will define an algebra of pattern matching automata. Elements of this algebra will be automata and defined operations will correspond to the creation of pattern matching problems.

Melichar and Holub also deal with pattern matching problems classification in their work [MH97]. However, their approach is different. They describe pattern matching problems using 6 criteria and therefore they can locate them in 6D space. They show how to construct a finite automaton for each point of the space. It is possible to start from a simple automaton which performs an exact match and transform it into a more sophisticated one. This inspired us to create an algebra of finite automata and to define operations with them which would correspond to transformations resulting in different points of the 6D space. This may provide an answer to an interesting question: will the number of obtained points be finite after a multiple application of such operations or not?

We wanted to construct operations for each of the six axes of the space which would generate automata corresponding to the points already identified. Then we could apply these operations several times and identify the properties of the algebra defined by the set of finite automata and these operations. In this paper we present up-to-date results of our work. We created an algebra of pattern matching automata in which only several of the intended operations are defined. To support our theory, we created a program in which we implemented operations identified on the automata so far. Using the program we obtained experimental data which was useful for us when constructing proofs for our statements.

In the next part we will give a precise definition of the algebra of pattern matching automata. Following that, we will characterize the algebra using identities discovered. In chapter 3 we will describe the program used during our experiment. To conclude with, we will mention other possible implementations of finite automata which are more suitable for practical application and outline our further research goals.

2 Definition of the algebra of pattern matching automata

2.1 Notation

Notation not listed directly in this paper can be found in [MI91].

Definition: Nondeterministic automaton is the quantiple M defined as $M = (\Sigma, Q, \delta, q_0, F)$, where

Σ is a finite alphabet

Q is a finite set of states

$q_0 \in Q$ is the start state

$F \subset Q$ is a set of final states

δ, a transition relation, is a finite subset of $Q \times \Sigma^* \times Q$.

In the remaining part of the text we will use an extended alphabet Σ', which is derived from Σ as follows:

Let Σ be a finite set. Then we can use Σ^- to denote the set $\{\bar{x} | \bar{x} = \Sigma - \{x\} \forall x \in \Sigma\}$. Let's use Γ to denote the symbol corresponding to the whole set and ε to denote the symbol corresponding to an empty transition. Then we can define the extended alphabet Σ' over the set Σ as $\Sigma \cup \Sigma^- \cup \{\varepsilon\} \cup \{\Gamma\}$.

For the definition of the algebra of pattern matching automata we will need to define an operation $\oplus$: merger of automata.

Definition: Let A be a set of automata using the same alphabet Σ and the same start state q_0. Then for the automata $X = (\Sigma, Q^X, \delta^X, q_0, F^X) \in A$ and $Y = (\Sigma, Q^Y, \delta^Y, q_0, F^Y) \in A$ we will define the automaton $X \oplus Y$ in the following way:

$$X \oplus Y = (\Sigma, Q^Y \cup Q^X, \delta^X \cup \delta^Y, q_0, F^X \cup F^Y)$$

Note: $\delta^X \cup \delta^Y$ for our purposes means a union of relations.

In the following we will assume that set A is closed with respect to construction $\oplus$. Then $(A, \oplus)$ is an algebra with a binary operation.

Theorem 1. Algebra $(A, \oplus)$ satisfies the following identities:

$$\begin{gathered} a \oplus a = a \\ a \oplus b = b \oplus a \\ (a \oplus b) \oplus c = a \oplus (b \oplus c) \\ \forall a, b, c \in A \end{gathered}$$

P r o o f.
The proof is obvious and ensues directly from the construction of operation $\oplus$. □

2.2 Definition of operations R, I, D

[MH97] introduced constructions $R(X, k)$ and $DIR(X, k)$.
These constructions correspond to the creation of a nondeterministic automaton X' from the automaton X which accepts a string $P = p_1 p_2 \dots p_n$. Automaton X' accepts only those strings P' with the value of P to P' distance equivalent to k while using distance R or DIR. Distance $R(P, P')$ is called Hamming distance and is defined as the minimum number of symbol replacement operations in string P required for the conversion of string P into string P'. Distance $DIR(P, P')$ is called Levenshtein distance and is defined as the minimum number of operations of symbol deletion (D), insertion (I) or replacement (R) in P required for the conversion of string P into string P'.
Automaton X' is called $R-$trellis or $DIR-$trellis as the case may be. Their construction is described, for example, in [MH97] or [HO96] (see Fig. 1).

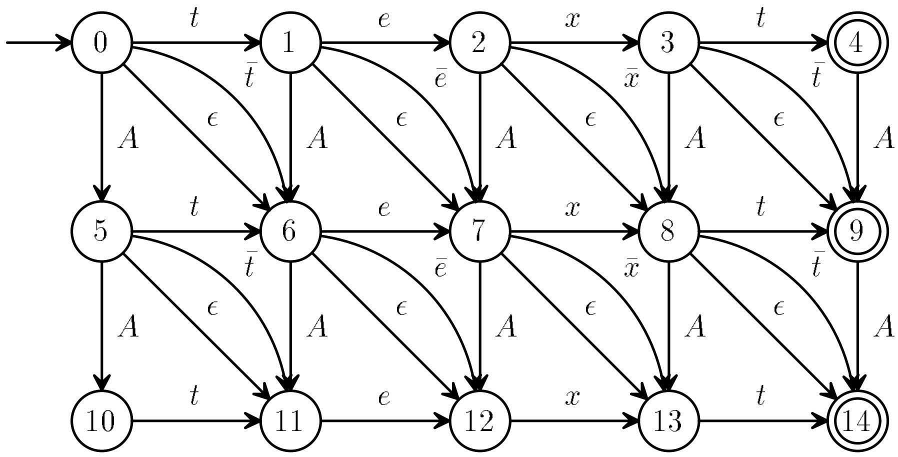


Figure 1: $DIR-$trellis for string “text”, $k = 2$.

When defining the operations in our algebra we are not restricted to the set of automata which perform an exact match but we define operations R and DIR for any finite automaton X in a similar way as Mužátko in his generalization of regular expression matching automata in [MU96]. Since distance DIR corresponds to any combination of operations of deletion D, insertion I, or replacement R, it is possible to define each of these operations independently.

Definition: Let $X = (\Sigma, Q, \delta, q_0, F)$ be a finite automaton and let $k \in N$. The result of operations $D(X, k)$, $I(X, k)$ or $R(X, k)$ is an automaton X' which will be derived from automaton X through the following steps:

1. Automaton X' will contain $k + 1$ clones of automaton X.

2. The states of automaton X' will be labelled $q_{i,j}$ where i is the sequence number of the clone and j is the sequence number of the state inside the original automaton X.

3. All transitions defined in the original automaton X will remain included in all its clones.

4. Error transitions will be added into automaton X' according to one of the following operations:

   Operation R: $\delta(q_{i,j}, \bar{a}) = \delta(q_{i+1,j}, a)$ shall be defined for each state $q_{i,j}(0 \leq i \leq k - 1, 0 \leq j \leq m-1)$ and for each symbol $a \in \Sigma$ for which transition $\delta(q_{i,j}, a)$ is defined. The symbol $\bar{a} \in \Sigma^{-}$ represents all symbols from alphabet Σ not equal to symbol a; or

   Operation D: $\delta(q_{i,j}, \varepsilon) = \delta(q_{i+1,j}, a)$ shall be defined for each state $q_{i,j}$ and for each symbol $a \in \Sigma, (0 \leq i \leq k - 1, 0 \leq j \leq m-1)$; or

   Operation I: $\delta(q_{i,j}, \Gamma) = q_{i+1,j}$ shall be defined for each state $q_{i,j}, (0 \leq i \leq k - 1, 0 \leq j \leq m-1)$.

5. Start state of automaton X' is state $q_{0,0}$. The alphabet of the automaton is the extended alphabet Σ'. The set of final states is the union of final states in all the clones: $F = F_0 \cup F_1 \cup \ldots F \cup F_k$.

Definition: Operations $D(X), R(X)$ and $I(X)$ correspond to operations $D(X,1), R(X,1)$ and $I(X,1)$ respectively.

Now we are ready to define the algebra of pattern matching automata.

Definition: The algebra of pattern matching automata is the algebra $\mathcal{A} = (A, D, I, R, *, +, \cdot, \oplus)$, where

A is a set of finite automata

$\oplus$ is an operation of merger

D is an operation of deletion

I is an operation of insertion

R is an operation of replacement

$*$ is an operation of closure

$+$ is an operation of union

$\cdot$ is an operation of concatenation

2.3 Properties of the algebra of pattern matching automata

Theorem 2. Let A be an algebra of pattern matching automata. Then for each automaton $X \in A$ it holds that

$$R(X,k) = R^k(X)$$

where $R^k(X)$ means $\underbrace{R(R(R(...R(X))...)))}_{k\ times}$.

`Proof.`

We will prove the theorem using mathematical induction.

1. According to the definition it holds: $R(X,1) = R(X)$.

2. Let's assume that $R(X,k) = R^k(X)$ holds.

3. If string $w_0 r_0 w_1 r_1 \dots w_k r_k w_{k+1}$ is accepted by automaton X, then automaton $R(X, k+1)$ accepts the string $w = w_0 r'_0 w_1 r'_1 \dots w_k r'_k w_{k+1}$, where $r'_i \in \{\bar{r}_i, r_i\}$. According to induction hypothesis, automaton R^k accepts string $w_0 r'_0 w_1 r'_1 \dots w_k r_k w_{k+1}$. After reading string $w_0 r'_0 w_1 r'_1 \dots w_k$, automaton $R^k(X)$ is in a state $q_{w_0 r'_0 w_1 r'_1 \dots w_k}$. It is obvious from the construction of automaton $R(R^k(X))$ that if $r'_k = \bar{r}_k$, then there is a transition from state $q_{w_0 r'_0 w_1 r'_1 \dots w_k, 0}$ into state $q_{w_0 r'_0 w_1 r'_1 \dots w_k r'_k, 1}$ and hence automaton $R(R^k(X)) = R^{k+1}(X)$ accepts string w. If $r'_k = r_k$, it is obvious that automaton $R(R^k(X)) = R^{k+1}(X)$ accepts string w.

   In reverse, let's assume that automaton $R^{k+1}(X)$ accepts string $w = w_0 r'_0 w_1 r'_1 \dots w_k r'_k w_{k+1}$. Then again according induction hypothesis it holds that $R^{k+1}(X) = R(R(X,k))$. Automaton $R(X,k)$ accepts string $w_0 r'_0 w_1 r'_1 \dots w_k r_k w_{k+1}$. It is obvious from the construction of the trellis that automaton $R(R(X,k))$ accepts string $w_0 r'_0 w_1 r'_1 \dots w_k r'_k w_{k+1}$ and hence the languages accepted by both automata, $R^{k+1}(X)$ and $R(X, k+1)$, are equal. □

Similar proofs can be provided for the remaining operations, D and I. Figure 2 shows an example of an automaton for operation D and $k = 2$.

We have not defined an equivalent of the DIR construction in our algebra. It is not really necessary. The corresponding DIR operation in our algebra will result from suitable application of operation $\oplus$ to automata $D(X), I(X)$ and $R(X)$:

$$DIR(X) = D(X) \oplus I(X) \oplus R(X)$$

The correctness of such a definition ensues from the behaviour of operation $\oplus$.

It is also possible to define the following in a similar way:

$$DI(X) = D(X) \oplus I(X)$$

$$DR(X) = D(X) \oplus R(X)$$

$$RI(X) = R(X) \oplus I(X)$$

Theorem 3. For any automaton $X = (\Sigma, Q, \delta, q_0, F)$ it holds that

$$DIR(X,k) = D^k(X) \oplus I^k(X) \oplus R^k(X)$$

Proof.
Let Σ' be an extended alphabet over alphabet Σ.
$DIR(X,1) = D(X) \oplus I(X) \oplus R(X)$ ensues directly from the definition of the construction.
If string $w = w_0 r_0 w_1 r_1 \dots w_k r_k w_{k+1}$, where $r_i \in \Sigma$, is accepted by automaton X, then automaton $DIR(X,k)$ accepts string $w = w_0 r'_0 w_1 r'_1 \dots w_k r'_k w_{k+1}$, where $r'_i \in \Sigma'$. If $r'_0 = \Gamma$, then having read string $w_0 r'_0$ using transitions from automaton $I^k(X)$, automaton $DIR(X,k)$ reaches state $q_{w_0 r'_0}$ which corresponds to the state of automaton X after the acceptance of word $w_0 r_0$. Similarly, it is possible to demonstrate that for any segment of string w', automaton $D^k(X) \oplus I^k(X) \oplus R^k(X)$ reaches the state

corresponding to the equivalent segment of string w.
It means that string w' transfers the automaton into a final state.
Let's suppose that string w is accepted by automaton X. The string accepted by automaton $D^k(X) \oplus I^k(X) \oplus R^k(X)$ will contain $k+1$ segments separated by symbols from set Σ'. It means that this string will be accepted by automaton $DIR(X,k)$ too.

□

3 Program

During the phase of building the hypotheses we carried out some experiments using our own program written in Microsoft Visual Basic 5.0. In this program we implemented nondeterministic finite automata and operations with such automata.

We used the method of simulation of the nondeterministic finite automaton in a deterministic way. The usage of symbols from the extended alphabet over the original alphabet prevented the rapid increase of states and transitions which would otherwise become inevitable during multiple application of the operation to the original string searching automaton.
The aim of the implementation was to verify our hypotheses and which was why we didn't pay any special attention to the effectiveness of the implementation. In case where the speed of the algorithm is one of the main criteria, it is possible to use a different type of implementation as described, for example by Mohri in [MM95].

In the following we give an example of pseudocode which demonstrates the algorithm of determination of active states after reading a terminal from input.

```
EpsilonPath(states)
  i <- 1
  While <= states.Count Do
     For Each trans In m_Transitions[states[i],Epsilon]
        Union (states,trans)
     i <- i + 1

NewStates(old_states, t)
  new_states <- EmptySet
  For Each index In t
     For Each or_state In old_states
        For Each trans In m_Transitions[or_state,index]
           If trans <> EmptyState And trans <> PreStartState Then
              Union (new_states,trans)
  EpsilonPath (new_states)
  NewStates <- new_states

Go(t)
  t_col <- m_TableOfTerminals.TermToIndexes(t)
  m_ActiveStates <- NewStates (m_ActiveStates, t_col)
```

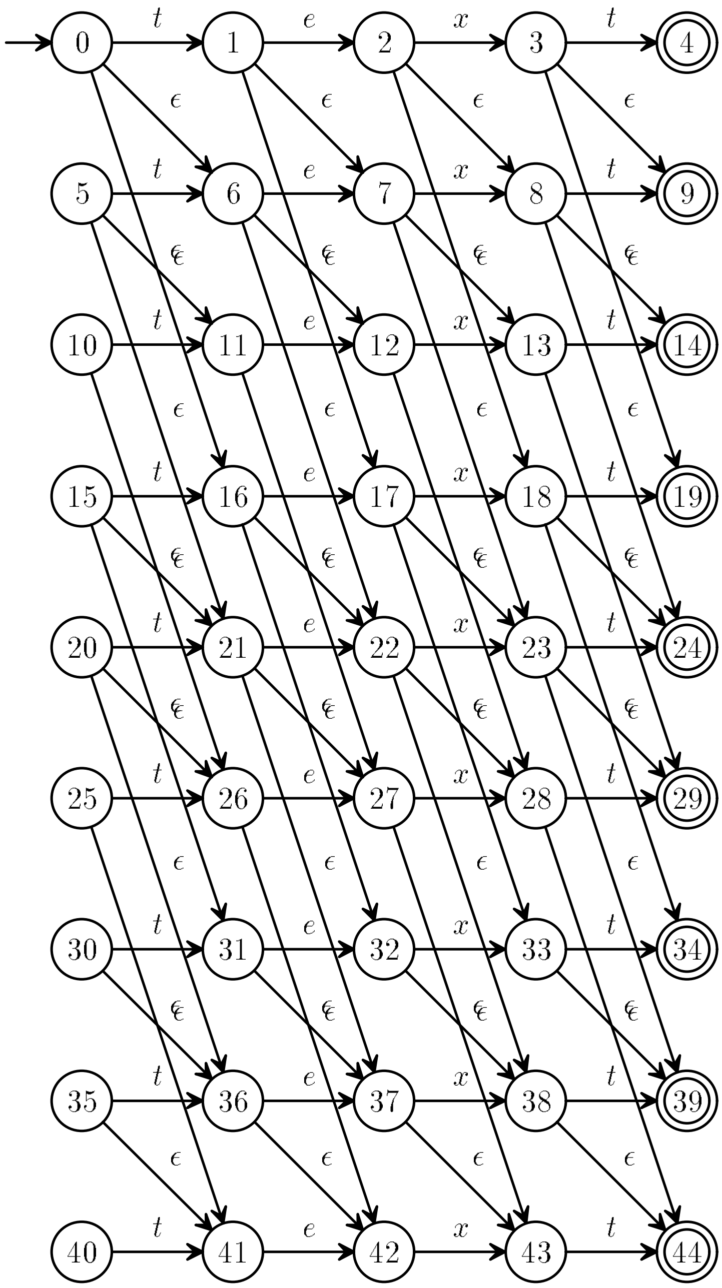


Figure 2: Double application of operation D on automaton performing exact match of string "text".

4 Conclusion

As we mentioned in the introduction we still find ourselves in the middle of the work. Our target is to create a description of pattern matching problems on an algebraic basis. This could be achieved by gradual addition of other operations which would correspond to the axes of the 6D space not included in our work yet. A further step of our research could be extension of the computing power of nondeterministic finite automata through the application of other models, such as multitape automata, and finding out whether they are suitable for pattern matching.

Another problem is the construction of an effective automaton (see [CR94]). As mentioned in the previous part of the paper, a construction scheme of the deterministic automaton for a regular expression can be found in [MM95]. We think that this scheme could be applicable for our purposes too.

References

[CR94] M. Crochemore, W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[HO96] J. Holub. *Reduced Nondeterministic Finite Automata for Approximate String Matching.* Proceedings of the Prague Stringology Club Workshop '96.

[LP81] H. R. Lewis, C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice Hall 1981.

[MH97] B. Melichar, J. Holub. *6D Classification of Pattern Matching Problems.* In this volume.

[MI91] B. Mikolajczak ed. *Algebraic and Structural Automata Theory.* North Holland 1991.

[MM95] M. Mohri. *Matching Patterns of an Automaton.* Combinatorial Pattern Matching, 6th Annual Symposium, CPM 95, Espoo, Finland, Springer Verlag 1995.

[MU96] P. Mužátko. *Approximate Regular Expression Matching.* Proceedings of the Prague Stringology Club Workshop '96.

[DP90] D. Perrin. *Finite Automata. Handbook of Theoretical Computer Science.* Elsevier Science Publishers 1990.