Collaborative Report DC–98–06

Proceedings
of the Prague Stringology Club Workshop '98

Edited by Jan Holub and Milan Šimánek

August 1998

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, Maxime Crochemore, Jan Holub, Bořivoj Melichar, Václav Snášel, Bruce W. Watson

Organizing Committee

Martin Bloch, Jan Holub, Martin Rýzl, Milan Šimánek, Zdeněk Troníček

Table of contents

Preface

This collaborative report contains the proceedings of the Prague Stringology Club Workshop '98 (PSCW'98), held at the Department of Computer Science and Engineering of Czech Technical University in Prague on September 3–4, 1998. The workshop was preceded by PSCW'96 which was the first action of the Prague Stringology Club and by PSCW'97. The proceedings of PSCW'96 and PSCW'97 were published as collaborative reports DC–96–10 and DC–97–03, respectively, 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 and PSCW'98 were selected from the papers submitted as a response to a call for papers. The papers in this proceedings are alphabetically ordered by the authors.

The PSCW aims at strengthening the connection between stringology (the 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 and Milan Šimánek, editors

A Fast Morphological Analysis Using the Extended AC Machine for Oriental Languages[1]

Kazuaki Ando, Kimihiro Iwasaki, Masao Fuketa and Jun-ichi Aoe

Department of Information Science & Intelligent Systems
University of Tokushima
2-1 Minami-Josanjima-Cho
Tokushima-Shi 770-8506
Japan

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

Abstract. This paper presents a fast morphological analysis for oriental languages by extending an Aho and Corasick's pattern matching machine. Our method is a simple and efficient algorithm to find all possible morphemes in an input sentence and in a single pass, and it stores the relations of grammatical connectivity of adjacent morphemes into the output functions. Therefore, the costs of checking connections between the adjacent morphemes can be reduced by using the connectivity relations. Furthermore, the construction method of the relations of grammatical connectivity is described. Finally, the proposed method is verified by a theoretical analysis, and an experimental estimation is supported by the computer simulation with a 100,267 words dictionary. From the simulation results, it turns out that the proposed method was 49.9% faster (CPU time) than the traditional trie approach. As for the number of candidates for checking connections, it was 25.5% less than that of the original morphological analysis.

Key words: morphological analysis, oriental language, dictionary lookup, trie structure, AC machine, grammatical connectivity

1 Introduction

An intelligent natural language interfaces enable users to communicate with the computer in English, Japanese or other human languages. Morphological analysis [ABE86, AKI94, KUR94, LEE97, MAR94, MOR96, SAN94] is the first step of natural language processing in the applications of natural language interfaces such as Information Retrieval [AOE91], Database Queries [KAP84], Expert Systems and so on. In general, the morphological analysis means segmentations of the input sentence into words (morphemes) and attachments of part-of-speech to them. Therefore, although morphological analysis for European languages, especially for English, plays only a minor role in a natural language processing system, in the analysis of oriental languages

[1]This work was supported by the Grant-in-Aid of the Ministry of Education, Science and Culture, Japan.

such as Japanese, Chinese and Korean it plays an important role because oriental languages are agglutinative languages, that is the language do not have explicit word boundaries between the words [ABE86, AKI94, KUR94, MAR94, MOR96, SAN94].

The procedure of morphological analysis of oriental languages consists of two steps. The first is to detect all possible morphemes, which are the smallest meaningful units, in a given input sentence. The second is to find the possible connections between adjacent morphemes by using a connection cost or probability based on the grammaticality [ABE86, AKI94, SAN94]. In the first step, the morphological analysis involves a large number of dictionary lookup. In general, a well-known technique for dictionary lookup is to use a trie structure [AOE91, AOE96, KUR94]. The trie is a tree structure in which each transition corresponds to a key character in the given keys set and common prefixes of keys can be shared. Therefore, the trie can search all keys made up from prefixes in an input string without the need of scanning the structure more than once. However, it is not so effective to use the trie for the morphological analysis [KUR94, MAR94, MOR96]. In order to detect all possible substrings in a given input sentence, the dictionary access must be tried repeatedly at each character position in the input sentence. Therefore, some characters may be scanned more than once for different starting positions and the number of dictionary accesses is increased. In the second step, the morphological analysis checks grammatical connectivity between adjacent words in order to find all possible connections [ABE86, SAN94]. This grammatical connectivity can be easily checked by using a grammatical table [ABE86]. However, this process requires considerable cost to check the grammatical connectivity, because it includes some checks of unnecessary connections, for example, checking connection between NOUN and CONJUGATION, since many words as part of speech have different grammatical interpretations. In order to achieve a fast morphological analysis, the mentioned problems should be solved.

This paper proposes a high speed morphological analysis of oriental languages by extending a pattern matching machine based on Aho and Corasick machine (called AC machine) [AHO75]. The proposed method is a simple and fast algorithm to find all possible substrings in an input sentence, and during only a single scan. Moreover, since the proposed method stores relations of grammatical connectivity of adjacent words into the output functions, the cost of checking connections between the adjacent words can be reduced by using the connectivity relations.

In the following sections, our ideas are described in detail. In Section 2, we describe the dictionary lookup method using a trie structure for the morphological analysis. Section 3 presents the high speed morphological analysis by extending the AC machine. Section 4 shows the theoretical analysis, and the experimental evaluations verified by the computer simulations with a 100,267 words dictionary. Finally, the results are summarized and the future research is discussed.

2 Dictionary Lookup Method using Trie in the Morphological Analysis

Morphological analysis of oriental languages is very different from that of English [ABE86, AKI94, KUR94, MAR94, MOR96, SAN94], because the languages do not have explicit word boundaries between the words as shown Fig. 1. Therefore, in order to find the most suitable word boundary, the morphological analysis must detect all possible substrings in a given input sentence in the first place. This process is one of the most important tasks of morphological analysis of oriental languages, since the wrong segmentation causes serious errors in the later analysis such as syntactic and semantic analysis [AKI94].

English : How lucky you are.

→ How / lucky / you / are / . /

Japanese : anatahanantekouunnanda = How lucky you are.

(a-na-ta-ha-na-n-te-ko-u-u-n-na-n-da)

→ a / na / ta / ha / na / n / te / ko / u / u / n / na / n / da /

ana / ta / ha / na / n / te / ko / u / u / n / na / n / da /

anata / ha / na / n / te / ko / u / u / n / na / n / da /

⋮

anata / ha / nante / kouun / na / n / da /

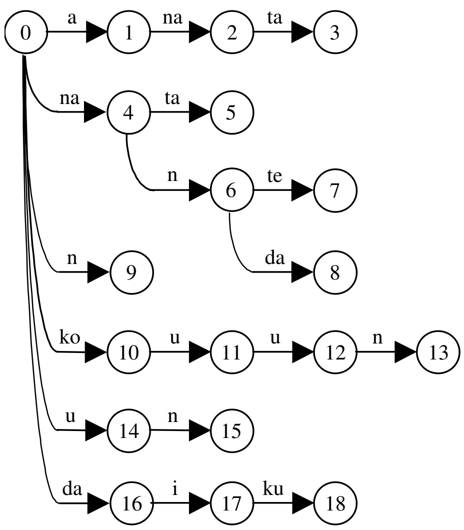
Figure 1: Difference in word boundary between English and Japanese.

In this process, the morphological analysis involves a large number of dictionary accesses. In general, a well-known method for dictionary lookup is to use a trie structure [AOE91, AOE96, KUR94]. The trie is a tree structure in which each transition corresponds to a character of the keys in the presented key set K. In the trie, a path from the root (initial state) to a leaf corresponds to one key in K. This way, the states of the trie correspond to the prefixes of keys in K.

The following is introduced for formal discussions:
1) S is a finite set of states, represented as a positive number.
2) I is a finite set of input symbols, or characters.
3) *goto* is a function from $S \times I$ to $S \cup \{fail\}$, called a goto function.
4) *output* is a function from S to morphological information, called an output function.
5) The state number of the trie is represented as a positive number, where the initial state in S is represented by the number 0.

A transition labeled with 'a' (in I) from r to n indicates that $goto(r,$ 'a'$) = n$. The absence of a transition indicates *failure* (*fail*). Fig. 2 shows an example of a trie for the set K = {"ana", "anata", "na", "nata", "nante", "nanda", "n", "ko", "kouu", "kouun", "u", "un", "da", "dai", "daiku"}. In this paper, for convenience of explanation, Japanese characters are described roman letters and a transition label is represented by the characters corresponding to the Japanese syllables. For example, retrieval of key "anata" is performed by traversing transitions $goto(0,$ 'a'$) = 1$, $goto(1,$ 'na'$) = 2$ and $goto(2,$ 'ta'$) = 3$, sequentially, and this time the key "ana"(=$output(2)$) and "anata"(=$output(3)$) are obtained.

In the morphological analysis, all possible substrings must be detected in order to find the most suitable word boundary. The following shows a dictionary lookup algorithm using a trie structure.



(a) The goto function.

s	*output*(*s*)
2	{ana} (hole)
3	{anata} (you)
4	{na}(name)
5	{nata}(hatchet)
7	{nante}(how)
8	{nanda}(what)
9	{n}(auxiliary verb)
10	{ko}(arc)
12	{kouu}(rainfall)
13	{kouun}(luck)
14	{u}(auxiliary verb)
15	{un}(destiny)
16	{da}(auxiliary verb)
17	{dai}(stand)
18	{daiku}(carpenter)

(b) The output function.

Figure 2: An example of TRIE.

Algorithm 1 : A dictionary lookup algorithm using a trie structure.
Input : A sentence $TEXT = c_1 c_2 ... c_n$, where each c_i, for $1 \leq i \leq n$, is an input character, a goto function *goto* and output function *output*.
Output : The morphological information of all possible substrings in a given input sentence *TEXT*.
Method :

Step 1-1 : { Initialization }
$i \leftarrow 1$;
Step 1-2 : { Change of the starting position }
$state \leftarrow 0$; $j \leftarrow i$;
Step 1-3 : { State transitions }
$state \leftarrow goto(state, c_j)$;
if $state = fail$ **then goto** Step 1-5;
if $output(state) \neq \phi$ **then print** $output(state)$;
Step 1-4 : { Operation control }
$j \leftarrow j + 1$;
if $j \leq n$ **then goto** Step 1-3;
Step 1-5 : { Operation control }
$i \leftarrow i + 1$;
if $i \leq n$ **then goto** Step 1-2;

The trie is a very common structure for dictionary access. However, it is not so effective to use the trie for morphological analysis of oriental languages, because the dictionary access must be tried from every character position in the input sentence, in order to detect all possible substrings in a given input sentence. Therefore, some characters may be scanned more than once for different starting positions and the number of unnecessary dictionary accesses is increased.

Consider the following input sentence (see Fig.2).

$TEXT$ = "kouunnanda (ko-u-u-n-na-n-da)" (It is lucky for me.)

The morphological analysis tries to find all possible words starting with "ko"(i=1). Then, the starting position is advanced to the second character in $TEXT$ and the dictionary access is repeated. The dictionary access is executed repeatedly until the end of input.

Step 1 : $i = 1$("ko"); The goto function fails at the character "na"; Keys "ko", "kouu" and "kouun" are found.
Step 2 : $i = 2$("u"); The goto function fails at the next character "u"; A key "u" is found.
Step 3 : $i = 3$("u"); The goto function fails at the character "na". Keys "u" and "un" are found.
Step 4 : $i = 4$("n"); The goto function fails at the character "na". A key "n" is found.
Step 5 : $i = 5$("na"); Keys "na" and "nanda" are found.
Step 6 : $i = 6$("n"); The goto function fails at the next character "da". A key "n" is found.
Step 7 : $i = 7$("da"); A key "da" is found and the process is finished.

As shown above, the character "u"(i=2), "u"(i=3), "n"(i=4), "na"(i=5), "n"(i=6) and "da"(i=7) were scanned two, three, three, four, two and three times, respectively, that is, the dictionary lookup was repeated 7 times.

3 Morphological Analysis Using the AC Machine

3.1 Dictionary Lookup

It was observed in the preceding section that morphological analysis using the trie involves a large number of dictionary accesses. In this section, in order to solve this problem, an efficient string pattern matching machine is used. Here a finite state string pattern matching machine based on the AC machine [AHO75, MAR94] locates all occurrences of any of a finite number of keywords in a text string.

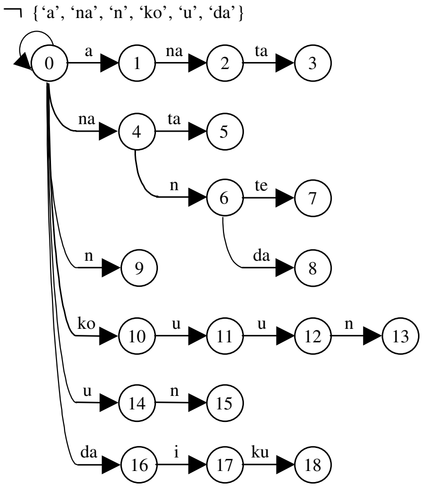
Let $KEY = \{k_1, k_2, ..., k_k\}$ be a finite set of strings which we shall call key and let $TEXT$ be an arbitrary string which we shall call the text string. The AC machine is a program which takes as input the text string $TEXT$ and produces as output the morphological information and the locations in $TEXT$ at which keys of KEY appear as substrings. The AC machine is constructed as a finite set of states S. Each state is represented by a number. One state (usually 0) is designated as the initial state.

The behavior of the AC machine is defined by the next three functions:

goto function $goto : S \times I \to S \cup \{fail\}$,
failure function $f : S \to S$,
output function $output : S \to A$, morphological information.

The function *goto* maps a set consisting of a state and a character into a state or the message *fail*. The function *f* maps a state into a state. The failure function is constructed whenever the goto function reports the message *fail*. Certain states are designated as output states which indicate that a set of keys has been found. The function output formalizes this concept by associating a set of keys (possible empty) with each state.

Fig. 3 shows the functions used by the AC machine for the set of keys $K = \{$"ana", "anata", "na", "nata", "nante", "nanda", "n", "ko", "kouu", "kouun", "u", "un", "da", "dai", "daiku"$\}$. Here, $\neg$ {'a','na','n','ko','u','da'} denotes all input characters other than 'a','na','n','ko','u','da'. The directed graph in Fig. 3(a) represents the goto function and the dotted line represents the failure function. For example, the transition labeled 'a' from state 0 to state 1 indicates that *goto*(0, 'a') = 1. The absence of transition indicates *fail*. The AC machine has the property that *goto*(0, 'σ') $\neq$ *fail* for all input symbols σ. A dictionary lookup algorithm using the AC machine is summarized below.



(a) The goto function.

s	*output*(*s*)
2	{ana, **na**} (hole, **name**)
3	{anata} (you)
4	{na}(name)
5	{nata}(hatchet)
6	{**n**}(**auxiliary verb**)
7	{nante}(how)
8	{nanda,**da**}(what,**auxiliary verb**)
9	{n}(auxiliary verb)
10	{ko}(arc)
11	{**u**}(**auxiliary verb**)
12	{kouu,**u**}(rainfall,**auxiliary verb**)
13	{kouun,**un**}(luck,**destiny**)
14	{u}(auxiliary verb)
15	{un,**n**}(destiny,**auxiliary verb**)
16	{da}(auxiliary verb)
17	{dai}(stand)
18	{daiku}(carpenter)

(b) The output function.

$f(2)=4, f(6)=9, f(8)=16, f(11)=14, f(12)=14, f(13)=15, f(15)=9,$
$f(0)=f(1)=f(3)=f(5)=f(7)=f(9)=f(10)=f(14)=f(16)=f(17)=f(18)=0;$

(c) The failre function.

Figure 3: An example of the AC machine.

Algorithm 2 : A dictionary lookup algorithm using the AC machine.
Input : A sentence $TEXT = c_1c_2...c_n$, where each c_i, for $1 \leq i \leq n$, is an input character, an AC machine with goto function *goto*, failure function *f*, and output function *output*.
Output : The morphological information of all possible substrings in a given input sentence $TEXT$.
Method :

Step 2-1 : { Initialization }
$state \leftarrow 0$;
$i \leftarrow 1$;

Step 2-2 : { State transitions }
if *goto*($state$, c_i) $\neq$ *fail* **then goto** Step 2-3;
$state \leftarrow$ *f(state)*;
goto Step 2-2;

Step 2-3 : { Output operation }

$state \leftarrow goto(state, c_i)$;
if $output(state) \neq \phi$ **then print** $output(state)$;
Step 2-4 : { Operation control }
$i \leftarrow i + 1$;
if $i \leq n$ **then goto** Step 2-2;

Consider the behavior of the AC machine that uses the functions in Fig. 3 to process the text string "kouun"(lucky). Since $goto(0, \text{'ko'}) = 10$, the AC machine enters state 10, advances to the next input symbol and emits $output(10)$, indicating that it has found the key "ko". Similarly, since $goto(10, \text{'u'}) = 11$, $goto(11, \text{'u'}) = 12$, and $goto(12, \text{'n'}) = 13$, the AC machine finds the $output(11)$(="u"), $output(12)$(="kouu" and "u") and $output(13)$("kouuni" and "un") respectively and enters state 13.

As shown above, the AC machine can find all possible substrings in an input sentence, scanned only once. Therefore, the AC machine is the most advantageous method for the morphological analysis.

3.2 Connection Check

In the second step of morphological analysis for oriental languages, the grammatical connectivities between adjacent words which were obtained by the dictionary lookup are checked in order to find the most suitable word boundary [ABE86, SAN94]. This grammatical connectivity can be easily checked by using a grammatical table which is described in a matrix of the connectivity between two words. However, this process requires considerable cost, because it must check the relation of all parts of speech which the preceding word and the following word have, and the unnecessary checks are included in those checks since many words have different kinds of parts of speech.

Let us now consider the Japanese words written in the syllabic alphabet called Hiragana. For example, suppose that Hiragana character 'ka' has 14 kinds of parts of speech and 'i' has 19 kinds of parts of speech in our dictionary for the morphological analysis as shown Figure 4. As for checks of grammatical connectivities between the preceding character 'ka' and the following character 'i' ("kai" means a shellfish, a floor etc.), it involves 266 (=14 $\times$ 19) kinds of checks. However, these checks includes 126 (=14 $\times$ 9) kinds of unnecessary checks such as checking a grammatical connectivity between NOUN and CONJUGATION. Such kinds of parts of speech, represented as bold types in Fig. 4, are called unconnection candidates. In other words, the unconnection candidate is a part of speech of the following word which is not connectable to all parts of speech of the preceding word. The problem of how to reduce the number of unnecessary checks should be solved in order to achieve a fast morphological analysis.

Thus, we extend one of the features of the AC machine in which information of substring are stored in one pass from initial state to terminal state by the failure function. By using this feature, the grammatical connectivities between the adjacent substrings which are included in one pass can be checked in advance, and the results can be stored into each output function as the unconnection candidate when the dictionary of morphological analysis is constructed. Therefore, if the unconnection candidate is available throughout the execution of the morphological analysis, the number of checking unnecessary connections can be reduced. For example, concerning

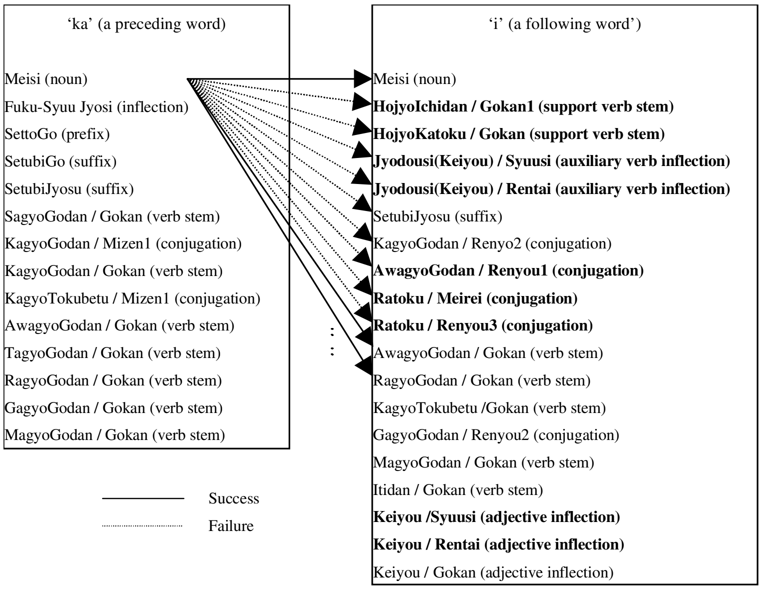


Figure 4: The connection checking between 'ka' and 'i'.

a pass "ta-be-su-gi" in Fig. 5, the relation between 'ta' and 'be', 'tabe' and 'su', and 'tabe' and 'sugi' can be checked in advance.

Consider the approaches using Directed Acyclic Word Graph (DAWG) or Finite State Automaton (FSA). In general, the DAWG and FSA have states with more than one outgoing transition after a state with two or more incoming transitions. Thus, there is no guarantee that there exists a subset of states in them with a one-to-one correspondence between the outputs and the states in that subset. Therefore, the DAWG and FSA cannot keep information of unconnection candidates correctly [AOE96]. But, by using the AC machine, information of unconnection candidates can be attached to the corresponding output state uniquely.

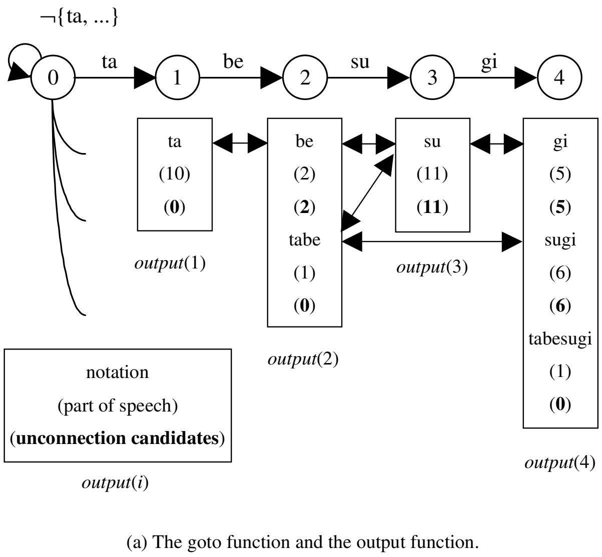


Figure 5: The connection checking for the pass "tabesugi".

Algorithm 3 summarizes the method for checking all unconnection candidates in

one pass, and the following variables and functions are utilized:
Variable *queue* : stores the states;
Variable *candidate* : stores a set of unconnection candidates returned by a function CCheck();
Function CCheck(*state1*, *state2*) : checks the grammatical connectivity between the adjacent words in *output*(*state1*) and *output*(*state2*) and it returns a set of unconnection candidates;
Function Update(*state*, *candidate*) : updates the *output*(*state*) on the basis of the *candidate*;

Algorithm 3 : Algorithm for checking all unconnection candidates in one pass.
Input : A key $KEY = c_1 c_2 ... c_n$ registered in the AC machine, where each c_i, for $1 \leq i \leq n$, is an input character and the AC machine.
Output : All unconnection candidates in one pass.
Method :

Step 3-1 : { Initialization }
$state \leftarrow 0$; $queue \leftarrow empty$; $i \leftarrow 1$;
Step 3-2 : { Storing states }
$state \leftarrow goto(state, c_i)$;
if $output(state) \neq empty$ **then** $queue \leftarrow queue \cup state$;
Step 3-3 : { Operation control }
$i \leftarrow i + 1$;
if $i \leq n$ **then goto** Step 3-2;
Step 3-4 : { Getting a state }
if $queue = empty$ **then** the process is terminated.
let *pre_state* be the next state in the *queue*;
$queue \leftarrow queue$ - $\{pre\_state\}$;
$tmp \leftarrow queue$;
Step 3-5 : { Connection checking }
if $tmp = empty$ **then goto** Step 3-6
let *next_state* be the next state in the *tmp*;
$tmp \leftarrow tmp$ - $\{next\_state\}$;
Un-Cand $\leftarrow$ ConnectionCheck(*pre_state*, *next_state*);
UpdateOutput(*next_state*, Un-Cand);
goto Step 3-5;
Step 3-6 : { Operation control }
goto Step 3-4;

Since all unconnection candidates in the dictionary can be checked by repeating the Algorithm 3 after the dictionary based on the AC machine is constructed, the number of checking unnecessary connections can be reduced by using the unconnection candidates.

In the example of the pass "ta-be-su-gi" in Fig. 5, when checking connections between the word starting with 'ta' and the following words, it involves $37 (= 10(\mathrm{ta}) \times 2(\mathrm{be}) + 1(\mathrm{tabe}) \times 11(\mathrm{su}) + 1(\mathrm{tabe}) \times 6(\mathrm{sugi}))$ kinds of checking by using our grammatical table. On the other hand, if the unconnection candidates are available, it is not necessary to check the grammatical connectivity, because all parts of speech of 'be', 'su' and 'sugi' are unconnection candidates, that is, it is $0(=10\times0+1\times0+1\times0)$. Therefore, the most suitable word boundary for the "tabesugi" is the only last position of the word, that is, "tabesugi/".

As shown the above, in some cases, the word boundary can be detected by using the unconnection candidate without checking the grammatical connectivity during the execution of the morphological analysis.

4 Evaluation

4.1 Theoretical Evaluation

Let n be the length of key. The precise complexity of algorithms presented depends on the data structures, so theoretical analysis is first discussed under the following assumptions:
1) The time complexity of confirming one transition, that is, a goto function is $O(1)$.
2) The time complexity of a failure function is $O(1)$.
3) The time complexity of a output function is $O(1)$.

Consider the time complexity of dictionary lookup. As for a trie, it is clear that the time complexity of retrieving a key is $O(n)$ [AOE91, AOE96]. However, in morphological analysis, since it must detect all possible substrings in a given input sentence, the number of the dictionary access depends on the length of the input sentence. Therefore, the time complexity becomes $O(n^2+n)(=O((n+1)n/2)$. On the other hand, the time complexity for dictionary lookup of the proposed method is $O(n)(=O(2n\text{-}1))$ [AHO75].

Next, consider the time complexity of construction. Suppose that k is the total number of length of the keys. Concerning the trie, the time complexity is $O(k)$ because it is proportional to the total length of keys. The time complexity for construction of the AC machine is $O(k)$ [AHO75]. The cost of the algorithm 3 depends on the function ConnectionCheck and the length of key. Let p be the average number of parts of speech of preceding words and let f be the average number of parts of speech of following words. Then, since the time complexity of the ConnectionCheck is $O(pf)$, the time complexity of the algorithm 3 becomes $O(n+pf(n^2\text{-}n))$ in the worst case, because the cost of for-loop in algorithm 3 is $O(n)$ and the cost of while-loop is $O((pf)(n\text{-}1)n/2)$. From above observation, the time complexity for constructing the proposed method becomes $O(s(l+pf(l^2\text{-}l)))$, where s is the total number of keys and l is the average length of keys.

Consider the effectiveness of the unconnection candidate. By using this candidate during the execution of the morphological analysis, the time complexity of the ConnectionCheck becomes $O(p(f\text{-}c))$, where c is the average number of the unconnection candidates which are stored in the output functions.

4.2 Experimental Evaluation

For experimental evaluation the following methods have been implemented on DELL OptiPlex XMT5120 (Pentium 120MHz) and they have been written in the C language.

(1) The dictionary lookup using a trie represented by a list structure.

(2) The proposed dictionary lookup method represented by a list structure.

In order to observe the effect of the proposed method, the following materials were used:

Dictionary : A 100,267 words dictionary of a Kana-to-Kanji translation system; The maximum length of keys is 19 and the average length of keys is 4.2. Note that Hiragana character in the Dictionary Source and the Input Text requires two bytes.

Inputs : News papers articles (2.7MByte); The total number of sentences is 39,339 and the average number of characters of the sentences is 72.6.

Table 1 shows the obtained simulation results. From these results, it turns out that the number of state transitions of the proposed method is 45.5% less than that of the trie method, and the proposed method is 49.9% faster (CPU time) than the trie one. However, the dictionary size of the proposed method is larger than that of a trie approach.

As for the candidates for checking connections, the total number of candidates is reduced from 5,637,007 to 4,014,625 by the unconnection candidates. This is 25.5% less than that of common morphological analysis. This means that the number of checking unnecessary connections can be reduced.

From the whole experimental observations, we can say that the proposed method is more practical than that of the trie. Although it requires more memory spaces, a fast morphological analysis can be achieved by using the proposed method.

Table 1: Simulation Results.

	Trie method	Our method
Dictionary Size (Mbyte)	8.3	32.5
State Transtions	14.444,170	8,016,514
Retrieval Time (sec.)	43.34	21.95

Conclusions

This paper has proposed a high speed morphological analysis by the AC machine. The proposed method is a simple and fast algorithm to find all possible substrings in an input sentence, and during a single scan, and it stores the connectivity relation of adjacent words into the output functions as the unconnection candidates. Therefore, if the unconnection candidates are available during the execution of the morphological analysis, the number of checking unnecessary connections can be reduced. Since these features depends on the passes, the proposed method have a good effect if there are a large number of long words in input text.

The efficiency of the proposed algorithm has been algorithm by the theoretical analysis, and the experimental evaluation was supported by the computer simulation with a 100,267 words dictionary. From the results, it turns out that the proposed method was 49.9% faster (CPU time) than the traditional trie approach. As for the number of candidates for checking connections, it was 28.8% less than that of the original morphological analysis by using the unconnection candidates.

As a future extension to this work, we are considering an implementation of morphological analysis system using the Multi-attribute AC machine [AND96] and the proposed method based on Double-Array Structure.

References

[ABE86] Abe, M. Ooshima, Y., Yuura, K., and Takeichi, N.: A Kana-Kanji Translation System for Non-Segmented Input Sentences Based on Syntactic and Semantic Analysis, Proceedings of the 10th International Conference on Computational Linguistics, 1986, pp.280-pp.285.

[AHO75] Aho, A.V., and Corasick, M.J.: Efficient String Matching : An Aid to Bibliographic Search, Communications of the ACM, Vol.18, No.6, 1975, pp.333-340.

[AKI94] Akiba, T., Tokunaga, T., and Tanaka, H.: An Extension of LangLAB for Japanese Morphological Analysis, Proceedings of the International Workshop on Sharable Natural Language Resources, 1994, pp.36-42.

[AND96] Ando, K., Shishibori, M., and Aoe, J.: An Efficient Multi-Attribute Pattern Matching Machine, Proceedings of the Prague Stringology Club Workshop'96, 1996, pp.1-14.

[AOE91] Aoe, J.: Computer Algorithms: Key Search Strategies, IEEE Computer Society Press, 1991.

[AOE96] Aoe, J., Morimoto, K., Shishibori, M., and Park, K.H.: A Trie Compaction Algorithm for a Large Set of Keys, IEEE Transactions of Knowledge and Date Engineering, Vol.8, No.3, June. 1996, pp.476-491.

[KAP84] Kaplan, S.J.: Designing a Portable Natural Language Database Query System, ACM Transactions on Database Systems, Vol.9, No.1, March.1984, pp.1-29.

[KUR94] Kurohashi, S., Nakamura, T., Matsumoto, Y., and Nagao, M.: Improvements of Japanese Morphological Analyzer JUMAN, Proceedings of the International Workshop on Sharable Natural Language Resources, 1994, pp.22-28.

[LEE97] Lee, G., Lee, J.H., B.-C., Kim, B.C., and Lee, Y.: A Viterbi-based morphological analysis for speech and natural language integration, Proceedings of the 17th International Conference on Computer Processing of Oriental Languages, Vol.1, 1997, pp.133-138.

[MAR94] Maruyama, H.: Backtracking-Free Dictionary Access Method for Japanese Morphological Analysis, Proceedings of the 15th International Conference on Computational Linguistics, 1994, pp.208-213.

[MOR96] Mori, S.: High Speed Morphological Analysis using DFA, Technical report of IEICE of Japan, NLC96-23, 1996, pp.17-23. (in Japanese)

[SAN94] Sano, H., Kawada, R., and Hasimoto, M.: Morphological Grammar Rules : An Implementation for JUMAN, Proceedings of the International Workshop on Sharable Natural Language Resources, 1994, pp.29-35.

The Longest Restricted Common Subsequence Problem[1]

Gabriela Andrejková

Department of Computer Science, College of Science,
P. J. Šafárik University,
Jesenná 5, 041 54 Košice, Slovakia

e-mail: `andrejk@kosice.upjs.sk`

Abstract. An efficient algorithm is presented that solves a Longest Restricted Common Subsequence Problem (RLCS) of two partitioned strings with the restricted using of elements. The above algorithm has an application in solution of the Set-Set Longest Common Subsequence Problem (SSLCS). It is shown the transformation of SSLCS Problem on RLCS Problem.

Key words: Design and analysis of algorithms, longest common subsequence, dynamic data structures.

1 Introduction

The common subsequence problem of two strings is to determine one of the subsequences that can be obtained by deleting zero or more symbols from each of the given strings.

The longest common subsequence problem (*LCS Problem*) of two strings is to determine the common subsequence with the maximal length.

For example, the strings *AGI* is a common subsequence and the string *ALGI* is the longest common subsequence of the strings *ALGORITHM* and *ALLEGATION*.

Algorithms for this problem can be used in chemical and genetic applications and in many problems concerning to the data and to the text processing. Genetic and chemical applications comprise the study of differences between long molecules such as proteins [14]. In the data processing and in the text processing the algorithms are used to determine an equivalence or a similarity of two strings [11] and to compress data when similar texts are being stored [4].

Further applications include the string-to-string correction problem [11] and determining the measure of differences between text files [4]. The length of the longest common subsequence (*LLCS Problem*) can determine the measure of differences (or similarities) of text files.

D. S. Hirschberg [6] presented $O(p \cdot n)$-time and $O(p \cdot (m-p) \cdot log\, n)$-time LCS algorithms, where m, n are the lengths of strings and p is the length of any longest common subsequence.

[1]This research was supported by Slovak Grant Agency for Science VEGA, Project No. 1/4375/97

J. W. Hunt and T. G. Szymanski [10] have presented $O((m+r) \cdot log\, n)$-time and $O(m+r)$-space algorithm, where m, n are lengths of strings and $r = |\{\langle i, j\rangle : a_i = b_j, 1 \le i \le m, 1 \le j \le n\}|$. G. Andrejková, Y. Robert and M. Tchuente [1, 15, 16] have presented systolic systems for LCS Problem with the combined complexity measures - $A \cdot T^2 = O(n^3)$ and $A \cdot P^2 = O(n^2)$, where A, T, P are complexity measures: area, time and period.

D. S. Hirschberg and L. L. Larmore [7] have discussed a generalization of LCS Problem, which is called Set LCS Problem (*SLCS Problem*) of two strings where however the strings are not of the same type. The first string is a sequence of the symbols and the second string is a sequence of subsets over an alphabet Ω. The elements of each subset can be used as an arbitrary permutation of elements in the subset. The longest common subsequence in this case is a sequence of symbols with maximal length. The SLCS Problem has an application to problems in computer driven music [7]. D. S. Hirschberg and L.L. Larmore have presented $O(m \cdot n)$-time and $O(m+n)$-space algorithm, m, n are lengths of strings.

The Set-Set LCS Problem (*SSLCS Problem*) is discussed by D. S. Hirschberg and L. L. Larmore [8] in 1989. In this case both strings are the strings of subsets over an alphabet Ω. In the paper is presented an $O(m \cdot n)$-time algorithm which solves the general SSLCS Problem.

In this paper we present an algorithm for special case of the LCS Problem, it means Longest Restricted Common Subsequence Problem (*LRCS Problem*) and its using to the solution of SSLCS Problem.

2 Basic Definitions

In this section, some basic definitions and results concerning to LRCS Problem and SSLCS Problem are presented.

Let Ω be a finite alphabet, $|\Omega| = s$, $P(\Omega)$ the set of all subsets of Ω, $|P(\Omega)| = 2^s$.

Let $A = a_1 a_2 \ldots a_m, a_i \in \Omega, 1 \le i \le m$ be a string over an alphabet Ω, $|A| = m$ is the length of the string A. A sequence of indices, $\mathrm{h}^{\mathrm{A}} = h_0^A h_1^A h_2^A \ldots h_{k^A}^A, 0 = h_0^A < h_1^A < h_2^A < \ldots < h_{k^A}^A = m, 1 \le k^A \le m$ is *a partition of the string A*.

The sequence h^{A} divides the string A in the following way:

$A = |a_1 a_2 \ldots a_{h_1^A}|a_{h_1^A+1} \ldots a_{h_2^A}| \ldots |a_{h_{k-1}^A+1} \ldots a_{h_{k^A}^A}| = subst_1^A \ldots subst_{k^A}^A$, where $subst_i^A = a_{h_{i-1}^A+1} \ldots a_{h_i^A}, 1 \le i \le k^A$. A pair $[A, h^A]$ is called *the string with the partition*. $\Omega(subst_r^A)$ is the alphabet of the substring $subst_r^A$.

For example, $\Omega = \{a, b, c, d, e\}$, $A = |abc|dababca|bd|daa|, m = 15$, $h^A = 0, 3, 10, 12, 15$; $subst_1^A = abc, subst_2^A = dababca, subst_3^A = bd, subst_4^A = daa$.

A string $C = c_1 c_2 \ldots c_p, 1 \le p \le m$ is *a restricted subsequence* of the string with the partition $[A, \mathrm{h}^{\mathrm{A}}]$, iff

1. there exists a sequence of indices $1 \le i_1 < i_2 < \ldots < i_p \le m$ such that $a_{i_t} = c_t, 1 \le t \le p$, and

2. if $h_{r-1}^A < i_u, i_v \le h_r^A$ then $c_u \ne c_v$, for all r, $1 \le r \le k^A$,

   (this means that each element of an alphabet $\Omega(subst_r^A)$ can be used in C once at most).

The string C is *a common restricted subsequence* of two strings with partition $[A, \mathrm{h}^{\mathrm{A}}]$ and $[B, \mathrm{h}^{\mathrm{B}}]$ if C is the restricted subsequence of $[A, \mathrm{h}^{\mathrm{A}}]$ and C is the restricted subsequence of $[B, \mathrm{h}^{\mathrm{B}}]$ at once. $|C|$ is the length of the restricted common subsequence.

The string C is *a longest common restricted subsequence* of two strings with partition $[A, \mathrm{h}^{\mathrm{A}}]$ and $[B, \mathrm{h}^{\mathrm{B}}]$ if C is a common restricted subsequence of the maximal length.

For example, $\Omega = \{a, b, c\}, A = |aba|abacac|bab|, m = 12, B = |babc|cac|cbcb|, n = 11$. The string $C = bacb$ is the restricted subsequence, $C' = bacab$ is the longest restricted common subsequence but the string $D = babccbb$ is not the restricted common subsequence for $[A, \mathrm{h}^{\mathrm{A}}]$ and $[B, \mathrm{h}^{\mathrm{B}}]$ as it can be seen in Figure 1. The string $C" = babcacbb$ is the longest common subsequence of the strings $A = abaabacacbab$ and $B = babccaccbcb$ if the partition does not matter.

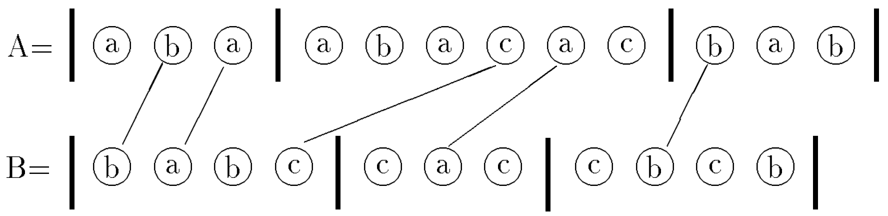


Figure 1. Restricted longest common subsequence of two strings A and B.

A string of sets $\mathcal{A}$ over an alphabet Ω is any finite sequence of sets from $P(\Omega)$. Formally, $\mathcal{A} = A_1A_2 \ldots A_m, A_i \in P(\Omega), 1 \leq i \leq m$, m is the number of sets in the string $\mathcal{A}$. The length of the symbol string described by $\mathcal{A}$ is $M = \Sigma_{i=1}^{m}|A_i|$.

A string of symbols $C = c_1c_2 \ldots c_p, c_i \in \Omega, 1 \leq i \leq p$, is *subsequence of symbols* (in short, a subsequence) of string $\mathcal{A}$ if there is nonincreasing mapping $F : \{1, 2, \ldots, p\} \rightarrow \{1, 2, \ldots, m\}$, such that

1. if $F(i) = k$ then $c_i \in A_k$, for $i = 1, 2, \ldots, p$
2. if $F(i) = k$ and $F(j) = k$ and $i \neq j$ then $c_i \neq c_j$.

Let $\mathcal{A} = A_1 \ldots A_m, \mathcal{B} = B_1 \ldots B_n, 1 \leq m \leq n$, be two strings of sets over the alphabet Ω. The string of symbols C is *a common subsequence of symbols* of $\mathcal{A}$ and $\mathcal{B}$ is C a subsequence of symbols of $\mathcal{A}$ and C is a subsequence of symbols of the string $\mathcal{B}$. *The longest common subsequence problem* of the strings $\mathcal{A}$ and $\mathcal{B}$ (SSLCS($\mathcal{A}, \mathcal{B}$) consists of finding a common subsequence of symbols C of the maximal length.

The length of SSLCS($\mathcal{A}, \mathcal{B}$) will be denoted $LSSLCS(\mathcal{A}, \mathcal{B})$. Note that C is not unique in general way.

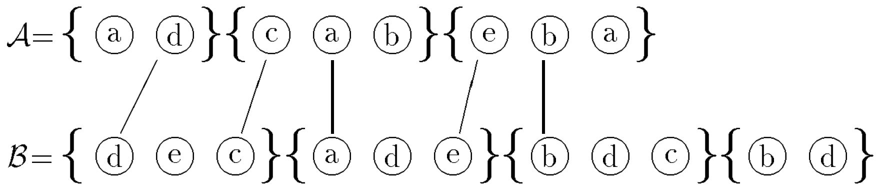


Figure 2. Longest common subsequence of two set strings $\mathcal{A}$ and $\mathcal{B}$.

For example, let $\Omega = \{a, b, c, d, e\}, \mathcal{A} = \{a, d\}\{a, b, c\}\{a, b, e\}, \mathcal{B} = \{c, d, e\}\{a, d, e\}$ $\{b, c, d\}\{b, d\}$. $C = abc$ is a common subsequence of symbols and $C' = adbcb$ and $C'' = dcaeb$ are the longest common subsequences of symbols for $\mathcal{A}, \mathcal{B}$. C'' can be seen in Figure 2.

3 Algorithm for LRCS Problem

Designation.

- $A[i..k] = a_i a_{i+1} \dots a_k$, for $1 \le i \le k \le m$,
- $\langle i, j\rangle$ represents i-th position in the string with the partition $[A, \mathrm{h^A}]$ and j-th position in $[B, \mathrm{h^B}]$, there exist indices r, s such that $1 \le r \le k^A, 1 \le s \le k^B$ and $h^A_{r-1} < i \le h^A_r, h^B_{s-1} < j \le h^B_s$,
- LRCS(A,B) is the longest restricted common subsequence of strings $[A, \mathrm{h^A}]$ and $[B, \mathrm{h^B}]$,
- $LLRCS(A, B)$ is the length of LRCS(A,B),
- $L(i, j) = LLRCS(A[1..i], B[1..j])$.

Principle of the recursive algorithm:

$LLRCS(A, B) = max_{|C|}\{|C| : C$ is the restricted common subsequence of $[A, h^A]$ and $[B, h^B]\}$.

Recursive version of the algorithm is constructed according to the following idea: If an element $c_t = a_{k_t} = b_{l_t}$ is in the $LRCS([A, h^A], [B, h^B])$ then

$$\begin{aligned} LLRCS([A, h^A], [B, h^B]) = 1 + {} & LLRCS([A[1..k_t - 1], h^{A'}], [B[1..l_t - 1], h^{B'}]) + \\ & LLRCS([A[k_t + 1..m], h^{A''}], [B[l_t + 1..n], h^{B''}]), \end{aligned}$$

where $h^{A'}, h^{A''}, h^{B'}, h^{B''}$ are partitions of the related substrings. The recursive version of the algorithm has the exponential time complexity.

A modified Hirschberg's method [6] will be used in the construction of the following time-polynomial algorithm.

A pair $\langle 0, 0\rangle$ is *a 0-candidate with an empty generating sequence.*

A pair of indices $\langle i, j\rangle, 1 \le i \le m, 1 \le j \le n, h^A_{r-1} < i \le h^A_r, h^B_{s-1} < j \le h^B_s$, will be named *a k-candidate*, $k \ge 1$, iff

1. $a_i = b_j$, and
2. there exists a sequence of pairs which is called *a generating sequence*: $\langle 0, 0\rangle = \langle i_0, j_0\rangle, \langle i_1, j_1\rangle, \dots, \langle i_{k-1}, j_{k-1}\rangle$ such that $i_{k-1} < i$ and $j_{k-1} < j$ and $\langle i_t, j_t\rangle$ is the t-candidate with the generating sequence $\langle i_0, j_0\rangle, \langle i_1, j_1\rangle, \dots,$ $\langle i_{t-1}, j_{t-1}\rangle$ and $(a_{i_t} \ne a_i$ or $(i_t \le h^A_{r-1})$ and $(b_{j_t} \ne b_j$ or $j_t \le h^B_{s-1})$ for $0 \le t \le k - 1$.

The set of all k-candidates will be designed $\mathcal{C}_k$ and the generating sequence of k-candidate will be designed I_{k-1}.

For example, $\langle 1,2\rangle, \langle 2,1\rangle, \langle 3,2\rangle, \langle 2,3\rangle, \langle 9,4\rangle, \langle 9,5\rangle, \langle 10,6\rangle \in \mathcal{C}_1$, $\langle 3,2\rangle, \langle 9,4\rangle, \langle 9,5\rangle, \langle 10,6\rangle \in \mathcal{C}_2$, $\langle 9,4\rangle, \langle 9,5\rangle, \langle 10,6\rangle \in \mathcal{C}_3, \langle 10,6\rangle \in \mathcal{C}_4, \dots$ for the strings with partitions $A = |aba|abacac|bab|$, $B = |babc|cac|cbcb|$.

Remark. $\langle i,j\rangle, h^A_{r-1} < i \leq h^A_r, h^B_{s-1} < j \leq h^B_s$ is 1-candidate with the generating sequence $\langle 0,0\rangle$ if $a_i = b_j$.

Lemma 3.1 *If the pair $\langle i,j\rangle, h^A_{r-1} < i \leq h^A_r, h^B_{s-1} < j \leq h^B_s$ is a k-candidate then $L(i,j) \geq k$.*

Proof. Let k=1 and $\langle i,j\rangle$ is 1-candidate with the generating sequence $\langle 0,0\rangle$. $a_i = b_j$, then $L(i,j) \geq 1$. Let $\langle i,j\rangle$ be a k-candidate. There exist two sequences of indices such that $i_1 < i_2 < \ldots < i_{k-1} < i$ and $j_1 < j_2 < \ldots < j_{k-1} < j$. $\langle i_{k-1}, j_{k-1}\rangle$ is $k-1$-candidate and we suppose $L(i_{k-1}, j_{k-1}) \geq k-1$. $a_i = b_j$ and $a_{i_t} = b_{j_t}$ for $1 \leq t \leq k-1$. The string $C = a_{i_1}a_{i_2}\ldots a_{i_{k-1}}a_i$ is the restricted common subsequence of $A[1..i]$ and $B[1..j]$ because of if $h^A_{r-1} < i_t, i_u \leq h^A_r$ then $a_{i_t} \neq a_{i_u}$ is fulfilled for all $r, 1 \leq r \leq k^A$. Analogously for $B[1..j]$. It follows that $L(i,j) \geq L(i_{k-1}, j_{k-1}) + 1 \geq k$. □

Lemma 3.2 *If $L(i,j) = k$ then there exists k-candidate $\langle i^*, j^*\rangle$ with the generating sequence I_{k-1} such that $i^* \leq i$ and $j^* \leq j$ and $L(i^*, j^*) = k$.*

Proof. If $L(i,j) = k$ then there is the restricted common subsequence $C = c_1c_2\ldots c_k$ which is created by elements in the positions determined by sequences $1 \leq i_1 < i_2 < \ldots < i_k \leq i, 1 \leq j_1 < j_2 < \ldots < j_k \leq j$, such that $a_{i_t} = c_t = b_{j_t}$ for $1 \leq t \leq k$, and from the definition of the restricted common subsequence follows:

1. if $h^A_{r-1} < i_u, i_v \leq h^A_r$, then $a_{i_u} \neq a_{i_v}$, for $1 \leq r \leq k^A$, and
2. if $h^B_{s-1} < j_u, j_v \leq h^B_s$, then $b_{j_u} \neq b_{j_v}$, for $1 \leq s \leq k^B$,

Let $i_u, i_v \in \{i_1, \ldots, i_k\}$. The 1. condition can be formulated as *not* $(h^A_{r-1} < i_u, i_v \leq h^A_r)$ or $a_{i_u} \neq a_{i_v}$. The first part means that i_u, i_v are not in the same interval of the partition h^A. If $i_u < i_v \leq h^A_r$ then $i_u \leq h^A_{r-1}$. The condition can be explained $a_{i_v} \neq a_{i_u}$ or $i_u \leq h^A_{r-1}$. Analogously for the condition 2.

Suppose that $i^* = i_k, j^* = j_k$ and $h^A_{r-1} < i_k \leq h^A_r, h^B_{s-1} < j_k \leq h^B_s$. The pair $\langle i_k, j_k\rangle$ is the k-candidate with the generating sequence $\langle 0,0\rangle, \langle i_1, j_1\rangle, \ldots, \langle i_{k-1}, j_{k-1}\rangle$, since $a_{i_k} = b_{j_k}$ and for all $t, 1 \leq t \leq k-1$, the pair $\langle i_t, j_t\rangle$ is the t-candidate with the generating subsequence I_{t-1} and $(a_{i_t} \neq a_{i_k}$ or $i_t \leq h^A_{r-1})$ and $(b_{j_t} \neq b_{j_k}$ or $j_t \leq h^B_{s-1})$. □

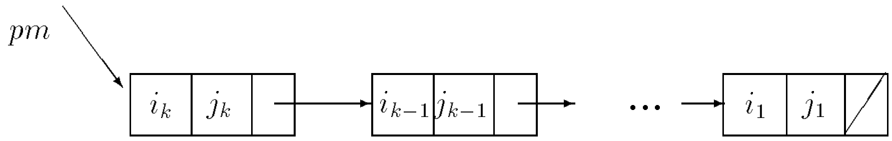
Lemma 3.3 *Let $C = c_1c_2\ldots c_k = a_{i_1}a_{i_2}\ldots a_{i_k} = b_{j_1}b_{j_2}\ldots b_{j_k}$ be the longest restricted common subsequence of $A[1..i]$ and $B[1..j]$ and $L(i,j) = k$ is its length. Let $h^A_{r-1} < i+1 \leq h^A_r, h^A_{s-1} < j+1 \leq h^B_s$. Let Cond A is the following condition:*

$a_{i+1} = b_{j+1}$ and $(a_{i+1} \neq a_{i_t}$ or $(i_t \leq h^A_{r-1}))$ and $(b_{j+1} \neq b_{j_t}$ or $(j_t \leq h^B_{s-1}))$ for all $t, 1 \leq t \leq k$.

If the Cond A is fulfilled then $\langle i+1, j+1\rangle$ is (k+1)-candidate and $L(i+1,j+1) = L(i,j)+1$, and the longest restricted common subsequence is $C^* = c_1c_2\ldots c_k a_{i+1}$. If Cond A is not fulfilled then L(i+1,j+1)=max $\{L(i,j+1), L(i+1,j)\}$ and the longest restricted common subsequence is in the same form as for max$\{L(i,j+1), L(i+1,j)\}$.

Proof. Suppose that *Cond A* is fulfilled. The sequence $\langle i_1, j_1\rangle, \ldots, \langle i_k, j_k\rangle$ is the generating sequence for (k+1)-candidate $\langle i+1, j+1\rangle$ since $i_k < i+1, j_k < j+1$, and for all $t, 1 \le t \le k$ the pair $\langle i_t, j_t\rangle$ is t-candidate with the generating subsequence I_{t-1} and ($a_{i_t} \neq a_{i+1}$ or $i_t \le h^A_{r-1}$) and ($b_{j+1} \neq b_{j_t}$ or $j_t \le h^B_{s-1}$). If assumptions of lemma are not fulfilled then $\langle i+1, j+1\rangle$ is not (k+1)-candidate and $L(i+1, j+1)$ can not be greater than $L(i, j+1)$ or $L(i+1, j)$. $\square$

Lemma 3.3 is the base for the construction of the algorithm for a computing of the restricted longest common subsequence of two strings with partitions. We use the dynamic data structure for the construction of linear lists representing the generating sequences of k-candidates, $k = 1, 2, \ldots$ as follows:



Algorithm will work with the following data types

```
{Omega is an alphabet of strings;}
type vertex = record            {element of generating sequence}
                x, y : integer; {indices}
                p: pointer;
             end;

type pointerv = ^vertex;        {pointer to the element of
                                  the generating sequence }

type  genseq = record           {record of the length and pointer}
                 length: integer;   {to the generating sequence}
                 pt: pointer;
              end;
```

The definition of the k-candidate gives the method for the construction of the k-candidate if the generating sequence is known. The next function *Candidate* finds if the element $\langle i, j\rangle$ is a potential k-candidate with a generating subsequence with pointer *pm*.

```
function Candidate(pm: pointer; ab: Omega; uA, uB: integer): Boolean;

{It returns the value "true" if <i,j> is a potential k-candidate
 else returns "false".

 pm - pointer to the generating subsequence,
 ab - the candidate in positions i, j,
 uA, uB - upper bounds of intervals for current positions
          i, j: uA<=i, uB<=j.}

var pp:pointerv; q: Boolean; ii, jj:integer;
```

```
begin
   pp:= pm; q:=true;
   while (pp<>nil) and q do
   begin
      ii:=pp^.x; jj:=pp^.y;
      if   (a[ii]=ab) and (ii>=uA)
        or (b[jj]=ab) and (jj>=uB) then q:=false;
      pp:= pp^.p
   end;
   Candidate:= q;
end; {Candidate}
```

Lemma 3.4 *The function* Candidate *computes the value* true *if* $\langle i, j\rangle$ *is a potential k-candidate else the value* false *in* $O(k)$*-time.*

Proof. pm is a pointer to the generating sequence of pairs $\langle i, j\rangle$, $\langle i, j\rangle$ is k-candidate. The function *Candidate* computes the value *false* if in this sequence there exists $\langle i^*, j^*\rangle$ such that $a_{i^*} = a_i = b_j$ and $i^* \geq uA$ or $b_{j^*} = a_i = b_j$ and $j^* \geq uB$. It means that the condition of k-candidate for $\langle i, j\rangle$ is not fulfilled. In the other case *Candidate* gives the value *true*, $\langle i, j\rangle$ is k-candidate with the given generating sequence. Time complexity is $O(k)$ because of each element of the generating sequence is compared with a_i k-times in the worst case. □

The function *Candidate* is used in the algorithm for computing a longest restricted common subsequence of two strings with some partitions.

ALGORITHM A:
{Algorithm constructs a longest restricted common subsequence of two strings with partitions.}

Input: $[A, hA], [B, hB]$ - two strings of symbols with partitions over alphabet $Omega$;
Output: pptr - pointer to the longest restricted common subsequence of A and B;
Variables:

Arrays C, D[0..m] of the type *genseq*.
C[i], D[i] - pointers to the longest common subsequences of A[1..i] and B[1..j];
$hA[1..kA], hB[1..kB]$ - arrays of partitions of the strings A and B;
uA, uB - upper bounds of intervals for current positions $i, j : uA \leq i, uB \leq j$.
dA, dB - the recent numbers of intervals in the partitions,
pp - a pointer to the vertex.

Method:

```
begin
   for i:=0  to n do
   begin
      D[i].pt:=nil; D[i].length:=0;
   end;
   C[0].pt:=nil; C[0].length:=0;
```

```
dA:=1; uA:=1;
for i:=1 to m do
   begin
      if i>hA[dA] then begin inc(dA); uA:=hA[dA-1]+1 end;
       dB:=1; uB:=1;
       for j:=1 to n do
       begin
          if j>hB[dB] then begin inc(dB); uB:=hB[dB-1]+1 end;
          if a[i]=b[j] then q:=Candidate(D[j-1].pt,a[i],uA,uB)
                       else q:=false;
          if q then
          begin
             new(pp);
             pp^.p:=D[i-1].pt; pp^.x:=i; pp^.y:=j;
             C[i].pt:=pp; C[i].length:=D[i-1].length+1;
          end  else
             if D[i].length>=C[i-1].length then C[i]:=D[i]
                                           else C[i]:=C[i-1];
        {Invariant1}
        end;
       for j:=1 to n do D[j]:=C[j];
       {Invariant2}
    end;
    len := C[n].length; pptr := C[n].pt;

{ "len" contains the length of the longest restricted common
  subsequence and C[n].pt contains pointer to the LRCS(A,B)}

          writeln('Length LRCS(A,B) =', len:3);
          while pptr<>nil do
          begin
               write(pptr^.x:3,pptr^.y:3,'**');
               pptr:=pptr^.p
          end;
end;
```

Theorem 3.1 *The Algorithm A computes correctly $LRCS(A,B)$ in $O(m \cdot n \cdot p)$-time and $O(n+r)$-space, where p is the length of $LRCS(A,B)$ and $r = |\{\langle i,j\rangle : a_i = b_j, 1 \le i \le m, 1 \le j \le n\}|$.*

Proof. We specify the invariants of the cycles in the algorithm A.
Invariant1:

$C[j']$ contains the length and the pointer to the $LCSS(A[1..i], B[1..j'])$, for $1 \le j' \le j$, and $C[j^*]$ contains the length and the pointer to the $LRCS(A[1..i-1], B[1..j^*])$ for $j < j^* \le n$.
Invariant2:

$C[j], D[j]$ contains the length and the pointer to the $LRCS(A[1..i], B[1..j])$ for $1 \le j \le n$ and $i \le n$.

The correctness of the algorithm follows immediately from the Invariant1 and Invariant2.

Time complexity: The function *Candidate* requires $O(k)$ steps, $k \leq p$, and it can be repeated at most $m \cdot n$ times. Thus, total time is $O(m \cdot n \cdot p)$.

Space complexity: The arrays C, D require $O(n)$ space, strings $[A, h^A]$ and $[B, h^B]$ require $O(m+n)$ space. If $a_i = b_j$ then function *Candidate* can give a value *true* and in this case a next element is added to the dynamic data structure that requires $O(r)$ space. If $m \leq n$ then the algorithm requires $O(n+r)$ space. $\square$

Let $\mathcal{C}_k$ be the set of all k-candidates, for some $k \geq 1$. *Partial ordering* "$\ll$" can be defined on $\mathcal{C}_k$ in the following way:
$\langle i, j\rangle \ll \langle i^*, j^*\rangle$ iff $i \leq i^*$ and $j \leq j^*$, for $\langle i, j\rangle, \langle i^*, j^*\rangle \in \mathcal{C}_k$.

An element $\langle i, j\rangle$ is a *minimal k-candidate* iff for all $\langle i^*, j^*\rangle \in \mathcal{C}_k, \langle i^*, j^*\rangle \neq \langle i, j\rangle$ is $i^* < i$ or $j^* < j$.

The set of all minimal k-candidates for $k \geq 1$, will be designed $\mathcal{C}_k^{min}$.

Remarks. It is clear that

1. $\mathcal{C}_1 \supseteq \mathcal{C}_2 \supseteq \ldots \supseteq \mathcal{C}_p \supseteq \mathcal{C}_{p+1} = \emptyset$
2. $\mathcal{C}_1^{min} \neq \mathcal{C}_2^{min} \neq \ldots \neq \mathcal{C}_p^{min}$
3. Let $1 \leq k \leq p, \langle i, j\rangle \in \mathcal{C}_k$ and $\langle i, j\rangle \notin \mathcal{C}_{k+1}$ then $L(i, j) = k$.

Hirschberg's method of minimal k-candidates [6] can be applied in this special case of strings with partitions and gives $O(n \cdot p^2)$-time algorithm, where p is the length of the longest restricted common subsequence.

4 Transformation of SSLCS Problem to LRSC Problem

Let $\mathcal{A} = A_1 A \ldots A_m, 1 \leq m$ be the string of the sets over Ω. Elements of a subset $A_i, A_i \in P(\Omega), 1 \leq i \leq m$, can be chosen in an arbitrary order and there are $|A_i|!$ permutations of these elements.

Let $p(A_i)$ be a permutation of elements inA_i (it is a string consisting of all symbols in A_i).

We define a string of symbols A in the following way:

$$A = p(A_1)p(A_2) \ldots p(A_m), \tag{1}$$

A is the concatenation of strings $p(A_1), p(A_2) \ldots, p(A_m)$.

Let $\mathbf{A}$ be the set of all strings of symbols created by (1). The number of elements in $\mathbf{A}$ is $|\mathbf{A}| = \Pi_{i=1}^{|A|} |A_i|!$. Let the elements in $\mathbf{A}$ are enumerated in some way, $\mathbf{A} = \{A^i\}, i = 1, \ldots, |\mathbf{A}|$.

Analogously, it is possible to construct the set $\mathbf{B}$ to the string $\mathcal{B}$. Let be

$$L(\mathbf{A}, \mathbf{B}) = max\, \{LLCS(A^i, B^j) : 1 \leq i \leq |\mathbf{A}|, 1 \leq j \leq |\mathbf{B}|\}. \tag{2}$$

Lemma 4.1 $L(\mathbf{A}, \mathbf{B}) = LSSLCS(\mathcal{A}, \mathcal{B})$.

Proof. Let $1 \leq i \leq |\mathbf{A}|, 1 \leq j \leq |\mathbf{B}|$. $LLCS(A^i, B^j)$ is the length of the longest common subsequence of strings of symbols A^i and B^j. Both strings are constructed as a special cases of strings $\mathcal{A}, \mathcal{B}$, and $LLCS(A^i, B^j) \leq LSSLCS(\mathcal{A}, \mathcal{B})$, for $1 \leq i \leq |\mathbf{A}|, 1 \leq j \leq |\mathbf{B}|$. It means $L(\mathbf{A}, \mathbf{B}) = max_{i,j}\{LLCS(A^i, B^j\} \leq LSSLCS(\mathcal{A}, \mathcal{B})$. Since all possible strings A^i and B^j have been used, the following inequality holds $L(\mathbf{A}, \mathbf{B}) \geq LSSLCS(\mathcal{A}, \mathcal{B})$. □

Let $1 \leq k \leq m$. $p^*(A_k)$ is constructed from $p(A_k)$ by adding some elements of A_k into arbitrary positions of $p(A_k)$. Each element of A_k is in the $p^*(A_k)$ once at least.

Lemma 4.2 *Let i, j be indices such that $L(\mathbf{A}, \mathbf{B}) = LLCS(A^i, B^j)$, $A^i = p(A_1)p(A_2) \dots p(A_m)$. Let $1 \leq k \leq m$ and $A^{i*} = p(A_1) \dots p(A_{k-1})p^*(A_k)p(A_{k+1}) \dots p(A_m)$. If each element of A_k can be chosen from $p^*(A_k)$ once at most then $L(\mathbf{A}, \mathbf{B}) = LLCS(A^{i*}, B^j)$.*

Proof. Since each element of A_k can be chosen from $p^*(A_k)$ once at most (some permutation of elements in A_k), we have $L(\mathbf{A}, \mathbf{B}) \geq LLCS(A^{i*}, B^j)$. $p^*(A_k)$ has been constructed by adding some elements to $p(A_k)$ and the following inequality is fulfilled: $LLCS(A^{i*}, B^j) \geq LLCS(A^i, B^j)$. □

Lemma 4.3 *Let i, j be indices such that $L(\mathbf{A}, \mathbf{B}) = LLCS(A^i, B^j)$. Let $A^{i*} = p^*(A_1) \dots p^*(A_m), B^{j*} = p^*(B_1) \dots p^*(B_n)$. If each element of $A_k, 1 \leq k \leq m$ can be chosen from $p^*(A_k)$ once at most and each element of $B_t, 1 \leq t \leq n$ can be chosen from $p^*(B_t)$ once at most then $L(\mathbf{A}, \mathbf{B}) = LLCS(A^{i*}, B^{j*})$.*

Proof. A^{i*} and B^{j*} are constructed by adding some elements to the strings A^i, B^j and thus $LLCS(A^{i*}, B^{j*}) \geq LLCS(A^i, B^j)$. Since each part $p^*(A_k)$, or $p^*(B_t)$ can be used as a permutation of A_k or B_t respectively, we have $L(\mathbf{A}, \mathbf{B}) \geq LLCS(A^i, B^j)$. Thus $L(\mathbf{A}, \mathbf{B}) = LLCS(A^i, B^j)$. □

Let $\mathcal{A} = A_1 A_2 \dots A_m, m \geq 1$ be the string of the sets over Ω. Let $p^+(A_k), 1 \leq k \leq m$ be the string of all permutations of A_k (permutations of elements in A_k are in $p^+(A_k)$ as the subsequences). Let $A^* = p^+(A_1)p^+(A_2) \dots p^+(A_m)$. Analogously for $\mathcal{B}$, $B^* = p^+(B_1)p^+(B_2) \dots p^+(B_n)$.

Theorem 4.1 $L(\mathbf{A}, \mathbf{B}) = LLCS(A^*, B^*)$ *if each element of A_k , respectively B_t, can be used once at most from the part $p^+(A_k)$, respectively $p^+(B_t)$.*

Proof. There are the indices i, j such that $L(\mathbf{A}, \mathbf{B}) = LLCS(A^i, B^j)$. According to Lemma 4.3 $LLCS(A^i, B^j) = LLCS(A^{i*}, B^{*j})$. The strings A^*, B^* are some special cases of strings A^{i*}, B^{j*} and it implies $L(\mathbf{A}, \mathbf{B}) = LLCS(A^*, B^*)$. □

Lemma 4.4 *The length of the string A^* is less or equal than M^2, the length of B^* is less or equal than N^2.*

Proof. $p+(A_k)$ can be constructed by a repeating of A_k $|A_k|$ times. This construction gives the length $|A_k|^2$. In [12] is presented the construction of shorter string with the length $|A_k|^2 - 2 \cdot |A_k| + 4, |A_k| \geq 4$. The length of A^* is $|A^*| = \Sigma_{k=1}^{m} |p^+(A_k)| \leq \Sigma_{k=1}^{m} |A_k|^2 \leq (\Sigma_{k=1}^{m} |A_k|)^2 = M^2$. Analogously, $|B^*| \leq N^2$. □

For example, let $\Omega = \{a, b, c, d, e\}, \mathcal{A} = \{a, d\}\{a, b, c\}\{a, b, e\}, \mathcal{B} = \{c, d, e\}\{a, d, e\}$ $\{b, c, d\}\{b, d\}$. It is possible to construct the following strings with partitions $[A^*, h^{A^*}]$ and $[B^*, h^{B^*}]$ to $\mathcal{A}$ and $\mathcal{B}$ respectively:

$A^* = |ada|cabcacb|ebaebea|, h^{A^*} = 0, 3, 10, 17, k^{A^*} = 3,$

$B^* = |decdedc|adeadae|bdcbdbc|bdb|, h^{B^*} = 0, 7, 14, 21, 24, k^{B^*} = 4.$

And the longest common subsequence of $\mathcal{A}$ and $\mathcal{B}$ can be computed by the algorithm for the restricted common subsequence problem of the strings with partitions $[A^*, h^{A^*}]$ and $[B^*, h^{B^*}]$: $LSSLCS(\mathcal{A}, \mathcal{B}) = LLRCS([A^*, h^{A^*}], [B^*, h^{B^*}])$.

Theorem 4.2 *Set-Set LCS Problem for two strings of sets can be computed in* $O(M^2 \cdot N^2 \cdot p)$ *time and* $O(N^2 + r)$ *space, where* M, N *are the numbers of symbols in subsets* $\mathcal{A}$ *or* $\mathcal{B}$*, respectively,* p *is the length of the longest common subsequence and* $r = |\{\langle i, j\rangle : a_i = b_j, a_i \in A^*, b_j \in B^*, 1 \leq i \leq M^2, 1 \leq j \leq N^2\}|$.

Proof. It follows from the Lemmas 4.2, 4.3, 4.4 and Theorem 4.1. □

5 Concluding Remarks

The polynomial algorithm for the solution of the LRCS Problem with a restricted using of elements has been presented. The algorithm can be used to show in the very simple way that SSLCS Problem has a polynomial complexity.

The LRCS Problem offers a generalization that is leading to the following problem: Let $[A, h^A], [B, h^B]$ be two strings with the partitions and with the restricted using of elements, let f_A, f_B are integer functions called weights of elements in positions: $f_A, f_B : \Omega \times \{1, 2, \ldots, n\} \to Integer$. For example, $A = abacbda$ the function f_A can have values $f_A(a, 3) = 7, f_A(a, 7) = 4, \ldots$. The measure of a common subsequence is the sum of weights of the matching elements. A weight of matching elements is the sum (or maximum) of weights of these elements in strings A and B in matching positions. Construct restricted common subsequence with the maximal measure.

References

[1] Andrejková, G.: *Systolic systems for the longest common subsequence problem.* Computers and Artificial Intelligence, 5 (1986), No. 3, p. 199–212.

[2] Apostolico, A.: *Improving the worst-case performance of the Hunt-Szymanski strategy for the longest common subsequence of two strings.* Information Processing Letters 23 (1986), p. 63–69.

[3] Dewar, R. B., Merritt, S. M., Sharir, M.: *Some modified algorithms for Dijkstra's longest common subsequence problem.* Acta Informatica 18, 1982, p. 1–15.

[4] Heckel, P.: *A technique for isolating differences between files.* Comm. ACM 21, 4 (Apr. 1978), p. 264–268.

[5] Hirschberg, D. S.: *A linear space algorithms for computing maximal common subsequences.* Comm. ACM 18, 6 (June 1975), p. 341–343.

[6] Hirschberg, D. S.: *Algorithms for longest common subsequence problem.* Journal ACM 24, 4 (Oct 1977), p. 664–675.

[7] Hirschberg, D. S., Larmore, L. L.: *The Set LCS Problem.* Algorithmica 2 (1987), p. 91–95.

[8] Hirschberg, D. S., Larmore, L. L.: *Set-Set LCS Problem.* Algorithmica 4 (1989), p. 503–510.

[9] Huang, S. S., Asuri, S. H.: *Algorithms for the Set-LCS and Set-Set-LCS Problems.* Tech. Report No. UH-CS-89-09, University of Houston, March, 1989.

[10] Hunt, J. W., Szymanski, T. G.: *A fast algorithm for computing longest common subsequences.* Comm. ACM 20, 5 (May 1977), p. 350–351.

[11] Lowrance, R., Wagner, R. A.: *An extension of the string-to-string correction problems.* Journal ACM 22, 2 (Apr. 1975), p. 177–183.

[12] Mohanty, S. P.: *Shortest string containing all permutations.* Discrete Mathematics 31, 1980, p. 91–95.

[13] Nakatsu, N., Kombayashi, Y., Yajima, S.: *A longest common subsequence algorithm suitable for similar text strings.* Acta Informatica 18, 1982, p. 171–179.

[14] Needleman, S. B., Wunsch, Ch. D.: *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal Mol. Biol. 48, 1970, p. 443–453.

[15] Robert, Y., Tchuante, M.: *A Systolic array for the longest common subsequence problem.* Information Processing Letters 21 (1985), p. 191–198.

[16] Robert, Y., Tchuante, M.: *Calcul en temps linéaire d'une plus longue sous-suite commune à deux chaîne sur une architecture systolique.* C. R. Acad. Sci. Paris, Série I, No. 7, 1984, p. 269–271.

Implementation of DAWG

Miroslav Balík

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

e-mail: `balikm@cslab.felk.cvut.cz`

Abstract. Let T be a text over a fixed alphabet A. Then an automaton can be created in a linear time that accepts all *substrings* that occur in text T. The ratio of the size of the implementation of this automaton *(factor automaton, DAWG)* and of the input text is in usual cases *14:1*. This paper shows a method of implementing *DAWG* that reduces this ratio down to *4:1* while preserving good qualities of the automaton, which is linear time of its construction with respect to the length of the input text and linear time of checking that a pattern is present in the text with respect to the length of the pattern.

Key words: finite automata, approximate string matching, DAWG, factor automaton

1 Introduction

Let $T = t_1t_2 \dots t_n$ be a *text* over a given alphabet A. An alphabet is a finite set of symbols. A *word* (string) over a given alphabet is a finite sequence of symbols. An empty sequence of symbols is called an empty word and it will be denoted as ε. A pattern $P = p_1p_2 \dots p_m$ is a substring of a text T iff such two natural numbers i, j exist that $P = t_it_{i+1} \dots t_j$. To answer whether a pattern P is a substring (subpattern, subword, factor) of a text T is a look-up problem.

A graph that represents a finite automaton accepting all substrings occurring in a given text is called *DAWG* (Directed Acyclic Word Graph).

The major advantages of *DAWG* are:

- it has a linear size limited by the number of vertices ,which is less than $2|n|-2$; the number of edges is less than $3|n|-4$, where $n > 1$ is the length of the text,
- it can be constructed in the time $\mathcal{O}(n)$,
- it allows to check whether a pattern occurs in a text in $\mathcal{O}(m)$, where m is the length of the pattern. Algorithm is shown on Fig. 1.

The basic idea of the implementation that is about to be described is that because the majority of edges contained in *DAWG* connect neighbouring vertices (according to a given topological order), these edges are worth implementing as a single bit saying whether such an edge is present or not. Another *DAWG* property is that all edges ending at such vertex are labeled by the same symbol of the alphabet, thus the labelling symbol can be transferred to vertices. Finally, when a statistical analysis of symbols and of the number of edges starting at a given vertex is performed, a suitable encoding can be employed to yield another reduction of *DAWG* size.

1. State $Q := Q_0$; $i := 1$;
2. ***if***$(i = m + 1)$**END** - Pattern occurs in Text
3. $Q := Successor(Q, P[i])$; $i := i + 1$
4. $if(Q = nil)$ **END** - Pattern does not occur in Text
   else goto(2)

Figure 1: Matching Algorithm

2 Implementation

The approach presented here creates a *DAWG* structure in three phases. The first phase is the construction of the usual *DAWG* graph, the second phase is topological ordering (or re-ordering) of vertices, which ensures that no edge has a negative "length", where length is measured as a difference of vertex numbers. The final phase is encoding and storing the resulting structure. More details about the implementation and the results presented here can be found in [Bal98].

2.1 Construction of *DAWG*

There are many ways of constructing *DAWG* from text, more details can be found for example in [Cro94]. The method used in this article is the on-line construction algorithm. An example of *DAWG* constructed using this algorithm for an input text $T = aabbabb$ is shown below:

During this phase a statistical distribution of symbols in the text is created. A statistical distribution of the number of edges at respective vertices is also created.

2.2 Topological Ordering

The *DAWG* structure is a directed acyclic graph. This means that its vertices can be ordered according to their interconnection by edges. Such an implementation that keeps all the information about edges starting from a vertex only in the vertex concerned while storing the vertices in a given order guarantees that every pattern matching will result in a single one-way pass through this structure.

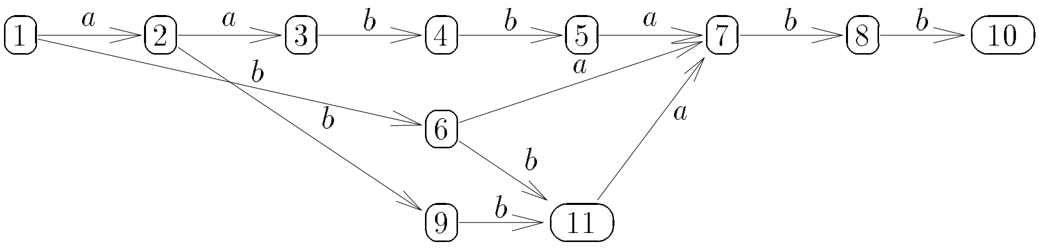


Figure 2: *DAWG* for the text $T = aabbabb$.

The problem of such topological ordering can be solved in linear time. At first, for each vertex its input degree (the number of edges ending at the vertex) is determined, next a list of vertices having an input degree equal to zero (the list of roots) is constructed. At the beginning, this list will contain only the initial vertex. One vertex is chosen from the list and for all vertices accessible by an edge starting at this vertex their input degree is decreased by one. Then such vertices that have a zero input degree are inserted into the list. And this goes on until the list is empty. The order of the vertices, which determines the quality of the final implementation, obtained this way depends on the strategy of choosing a vertex from the list. Several strategies were tested and the best results were obtained using the LIFO (last in - first out) strategy, for more details see [Bal98].

The original *DAWG* shown in Fig. 1 will be reordered using the LIFO strategy and the resulting graph is depicted in Fig. 3

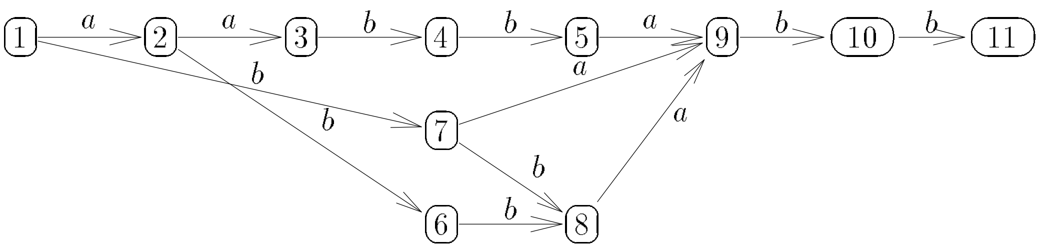


Figure 3: The result of reordering.

2.3 Encoding

The *DAWG* graph is encoded element by element (elements are described later in this section). It starts with the last vertex according to the topological order (as described above) and progresses in the reverse order, ending with the first vertex of the order. This ensures that a vertex position can be defined by the first bit of its representation and that all edges starting at the current vertex can be stored because all ending vertices have already been processed and their address is known.

The highest building block is a *graph*. It is further divided into single *elements*. Each *element* consists of two parts: a *vertex* and an *edge*. A *vertex* carries out an information on a label of all edges ending at it. A Huffman code is used for coding of the respective symbol of the alphabet. An *edge* is further split into a *header* and an *address order*. A *header* carries out information on the *number of addresses* - edges belonging to a respective vertex. A distribution of edge counts for all vertices can be obtained during the construction of *DAWG*. This makes possible to use a Huffman code for header encoding, but Fibonacci encoding is sufficient as well, though one must expect a substantial amount of small numbers. An *address* is the address of the first bit of the element being pointed to by an appropriate edge. It is further split to two parts, one describing the length of the other part, which is a binary encoded address.

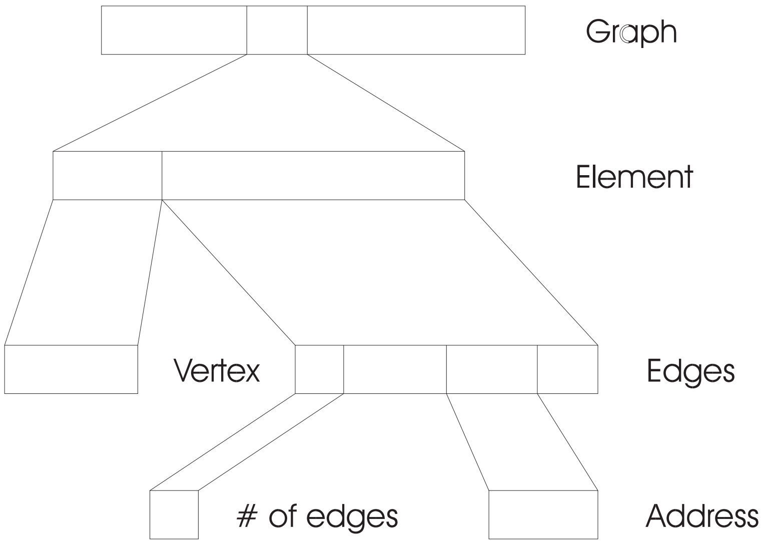


Figure 4: Implementation - Data Structures

2.4 Matching Algorithm

1. Build coding trees from the $CodeFile$
2. $Ptr := 0$; $\{Ptr \ldots$pointer into the $CodeFile\}$
   $i := 1$; initialize $Stack$
3. $\boldsymbol{if}(i = m + 1)$**END** - Pattern occurs in Text
4. Decode $num \ldots$number of edges starting in state Q, update Ptr.
5. $\boldsymbol{if}$ (num = "*Only one edge to the next vertex*") $Push(Stack, 0)$
   $\boldsymbol{else}$ $\boldsymbol{while}(num > 0)$ {decode one edge and push it to $Stack$; $num := num - 1$; update Ptr }
6. $\boldsymbol{if}$ ($Stack$ is empty)**END** - Pattern does not occur in Text
7. $Ptr := Ptr + Pop(Stack)$
8. Decode $label$ from Ptr
9. $\boldsymbol{if}$ ($label = P[i]$) { update Ptr; i := i+1; $\boldsymbol{goto}(3)$}
   $\boldsymbol{else}$ $\boldsymbol{goto}(6)$

2.5 Symbol Encoding

A code of an *element* (vertex and corresponding edges) starts with a code of the symbol for which it is possible to enter the vertex. The best code is the Huffman code, which can be based either on counts of symbol occurrences in the text, or proportionate representation at individual vertices. The latter better suits the implementation.

File Name	$\|X\|$	Symbol Count $\frac{\|Bits\|}{\|Symbol\|}$	Proportionate Repre. $\frac{\|Bits\|}{\|Symbol\|}$
TEXT1	21818	4.771186	4.770672
TEXT2	53801	4.264782	4.264746
TEXT3	81054	4.588081	4.587633
RANDOM1K	1000	7.532672	7.531844
RANDOM10K	10051	7.809383	7.807136
RANDOM100K	100447	7.831894	7.831680

The average number of bits necessary to store one symbol is calculated for symbols representing the vertices of the graph. It can be observed that the two methods of encoding provide similar results. For example, using the latter method for encoding the file TEXT3 will result in improvement of only 0.00045 bits per symbol, which is 0.0098% with respect to the value obtained using the first method.

2.6 Symbol Decoding

Decoding begins at the *root* of the coding tree, and follows a left edge when a '0' is read or a right edge when a '1' is read. When a leaf is encountered, the corresponding symbol is output.

2.7 Encoding of Number of Edges

The code of the *number of edges* is another item. Even this value can be obtained prior to encoding. A typical example of a distribution of numbers of edges for two input text files is shown in the Fig. 6.

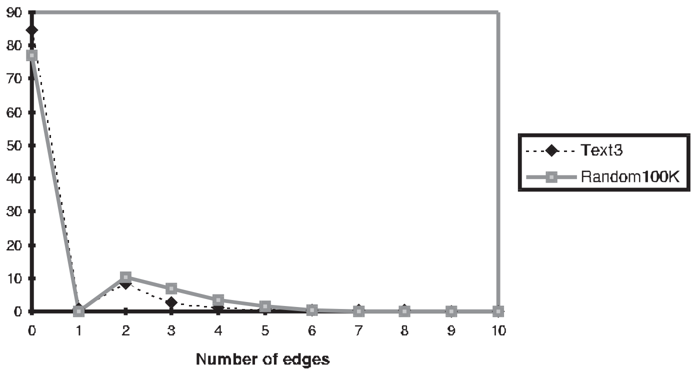


Figure 5: Edge count distribution (# of edges, # of vertices - %)

In the figure Fig. 5 vertices with just one edge starting at them were further divided into two groups: the first group is formed by vertices having just one edge leading to the next vertex according to given vertex ordering (included in the group Edge count = 0), and the second group is formed by vertices having just one edge leading anywhere else (Edge count = 1). The first group can be easily encoded by the value of Edge count

The figure also shows that more than 84% of all vertices belong to the first group. This means that the code word describing this fact should be very short. It will be only one bit long using Huffman coding. Other values of edge counts are represented by more bits according to the structure of the input text.

The smallest element of *DAWG* represents a vertex with just one edge ending at the next vertex. For TEXT3 it is 5.6 (4.6 per symbol + 1 bit per edges) bits on average. The fact that *DAWG* consists mainly of such elements was used in the construction of the *Compact DAWG structure (CDAWG)* derived from the general *DAWG*, more details can be found in [Cro97].

2.8 Number of Edges Decoding

The process is similar to Symbol decoding.

2.9 Edge Encoding

The last part of the graph element contains references to vertices that can be accessed from the current vertex. These references are realised as relative addresses with respect to the beginning of the next element. The valid values are non-negative numbers. To evaluate them it is necessary to know the ending positions of corresponding edges. This is why the code file is created by analysing *DAWG* from the last vertex towards the root in an order that excludes negative edges. If we wanted to work with these edges, we would have to reserve an address space to be filled in later when the position of the ending vertex is known.

The address space for a given edge depends on the number of bits representing the elements (vertices) lying between the starting and ending vertices. As the size of these elements is not fixed (the size of the dynamic part depends mainly on element addresses), it is impossible to obtain an exact statistical distribution of values of these addresses, which we obtained for symbols and edges. A poor implementation of these addresses will result in the fact that elements will be more distant and the value range broader.

Yet it is possible to make an estimation based on the distribution of edge lengths (measured by the number of vertices between the starting and ending vertices). In this case the real address value might be only $q - times$ higher on average, where q is an average length of one *DAWG* element. The first estimation of optimal address encoding is based on the fact that the number of addresses covered by k bits is the same for $k = 1, 2, ..., t$, where t is the number of bits of the maximum address. We will use an address consisting of two parts: the first part will determine the number of bits of the second part, the second part will determine the distance of the ending vertex in bits. The simplest case is when the addresses are of a fixed length, then the length of an average address field is $r = s + t$, where $s = 0$, which means that $r = t$

actually. Another significant case is a situation when the number of categories is t, then $s = \lceil \log_2 t \rceil$.

When s is chosen from an interval $s \in \langle 0, \lceil \log_2 t \rceil \rangle$, the number of categories is 2^s, the number of address bits of the $i-th$ category is $\frac{t*i}{2^s}$. An average address field length is then

$$r = s + \sum_{i=1}^{2^s} \frac{t * i}{2^s}.$$

When we rearrange this formula, we obtain

$$r = s + t\frac{2^s + 1}{2^{s+1}}.$$

When the address length is fixed and the number of categories varies, this function has a local minimum for

$$2^s = \frac{t \ln 2}{2}.$$

If we know t, we can calculate s as

$$s = \log_2(t \ln 2) - 1$$

s	t	Optimal $\|X\|$
1	6	3B
1 and 2	8	11B
2	12	171B
2 and 3	16	2.7kB
3	23	350kB
3 and 4	32	180MB
4	46	2.9TB
4 and 5	64	$8 \cdot 10^{17}$B
5	92	$2 \cdot 10^{26}$B

The above table shows optimal values of t for given values of s as well as address limits when it does not matter if we use a code for s or $s+1$ categories. The estimation of the input file length assumes that the code file is three times greater than is the length of the input text, and that the code file contains the longest possible edge, which connects the initial and the last vertices. This observation is based on experimental evaluation.

It can be seen that the value $s = 3$ is sufficient for a wide range of input text file lengths, which guarantees a simple implementation, yet it leaves some space for doubts about the quality of the approach used. Or is it so that edge lengths are not spread uniformly in the whole range of possible edge lengths (1 to the maximum length)? The answer can be found in the following figure.

The figure does not contain edges ending at the next vertex (with respect to the actual vertex) as they are dealt with in a different way. It can be clearly observed that the assumption of uniformity of the distribution is not quite fulfilled. Nevertheless categories can be constructed in the way that supports the requirement of the minimal average code word length. The other two figures depict the real distribution of address

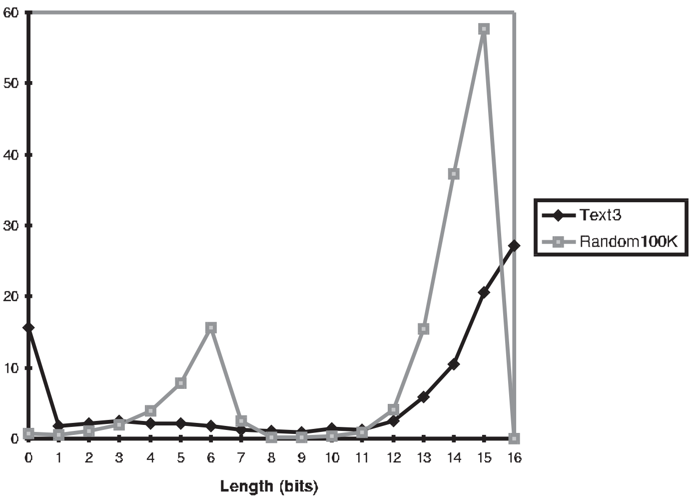


Figure 6: Edge Length Distribution (Length - bits, # of edges - %)

lengths for two ways of encoding. The first is a code with two categories, one encoding addresses with 15 bits, the other with 30 bits. The second way regularly divides address codes into eight categories by four bits.

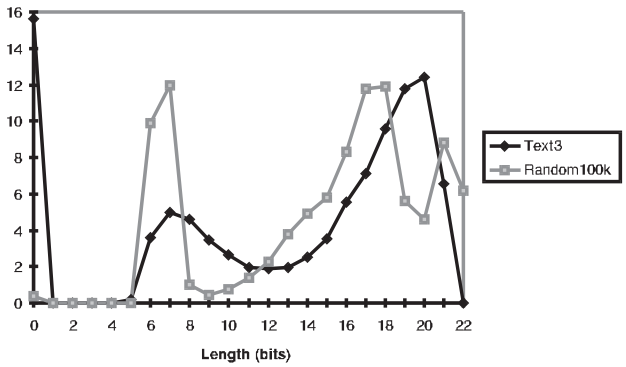


Figure 7: Address length distribution - Two categories (Length - bits, # of addresses - %)

Both ways of address encoding provide similar results. The relevancy with respect to the statistical distribution of edges is obvious, the peaks being shifted by three or four bits to the right.

2.10 Edge Decoding

Decoding depends on the number of categories used for encoding. When eight categories are used, three bits are used for symbol length code – $s = 3$. We read these three bits as an integer n. Then we calculate the number of bits that represent an edge address as $t := (n + 1) * const$, where $const$ is based on the length of $CodeFile$. Than, we read n bits from $CodeFile$ as an integer, and this number is the address.

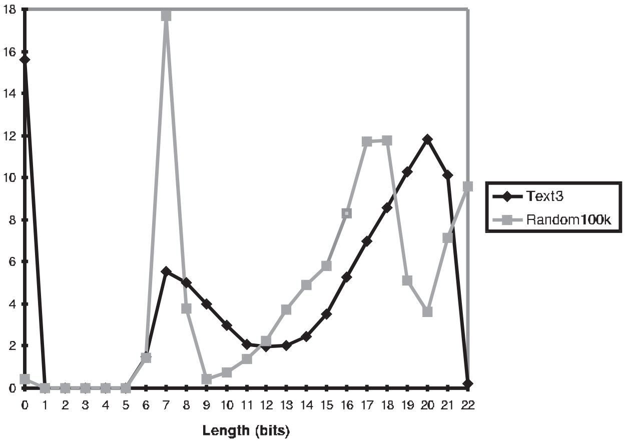


Figure 8: Address length distribution - Eight categories (Length - bits, # of addresses - %)

The following picture describes the contribution of individual parts to the overall length of the resulting code.

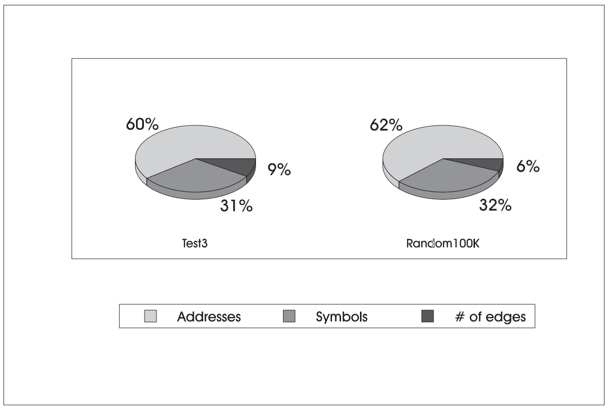


Figure 9: The influence of code lengths to the overall length of the code file

The biggest portion is occupied by edge encoding, even though the majority of edges is included in the edge count encoding. The test was performed for encoding with eight address categories.

2.11 Complexity

DAWG can be created using the on-line construction algorithm in $\mathcal{O}(n)$ time [Cro94]. Vertex re-ordering can be also done in $\mathcal{O}(n)$ time, encoding of *DAWG* elements as described above can also be done in $\mathcal{O}(n)$ [Bal98]. Moreover, vertex re-ordering can be done during the first or third phases. This means that the described *DAWG* construction can be performed in $\mathcal{O}(n)$.

The time complexity of searching in such an encoded *DAWG* is $\mathcal{O}(m)$ [Bal98].

3 Results

File Name	$\|\mathbf{X}\|$	$\|\mathbf{Y_1}\|$	$\|\mathbf{Y_2}\|$	$\frac{\|\mathbf{Y_1}\|}{\|\mathbf{X}\|}$	$\frac{\|\mathbf{Y_2}\|}{\|\mathbf{X}\|}$
TEXT1	21818	602928	500385	345.4 %	286.7 %
TEXT2	53801	1459973	1201342	339.2 %	279.1 %
TEXT3	81054	2304026	1906376	355.3 %	294.0 %
MOD4005.TXT	1246946	-	11818341	-	118.5 %
RANDOM1K	1000	25687	23258	321.1 %	290.7 %
RANDOM10K	10051	244703	219157	304.3 %	272.6 %
RANDOM100K	100447	3843810	3177465	478.3 %	395.4 %

The size of the code file for two sets of addresses is denoted as $|Y_1|$, $|Y_2|$ is relevant for the code using eight address categories. Both values are in bits and do not contain information on the Huffman encoding used. The size of these data does not depend on the size of the input file.

4 Conclusion

The results show that the ratio of code file size vs. the input file size is 3:1. This number changes very little with the rising size of the input file to the detriment of the code file. If the ratio rose as high as 4:1, a CD-ROM with the capacity of 600MB could contain one code file for an input file of the maximal size up to 150MB, which is a more than three-times better result than the one obtained by the classical approach.

References

[Ada89] J. Adamek: *Coding.* MVŠT XXXI, SNTL, Prague, 1989, in Czech.

[Bal98] M. Balík: *String Matching in a Text.* Diploma Thesis, CTU, Dept. of Computer Science & Engineering, Prague, 1998.

[Cro94] M.Crochemore, W.Rytter: *Text Algorithms*, Oxford University Press, New York, 1994.

[Cro97] M.Crochemore and R.Vérin: *Direct Construction Of Compact Directed Acyclic Word Graphs.* in (CPM97, A. Apostolico and J. Hein, eds., LNCS 1264, Springer–Verlag, 1997) pp 116-129.

[Me95] B. Melichar: *Approximate String Matching By Finite Automata.* Computer Analysis of Images and Patterns, LNCS 970, Springer, Berlin 1995.

[Me96] B. Melichar: *Fulltext Systems.* Publishing house CTU, Prague, 1996, in Czech.

[Me97] B. Melichar: *Pattern Matching and Finite Automata.* Proceedings of the Prague Stringology Club Workshop '97, Prague, 1997.

Exact String Matching Animation in Java[1]

Christian Charras and Thierry Lecroq

LIR (Laboratoire d'Informatique de Rouen) and
ABISS (Atelier Biologie Informatique Statistiques Socio-linguistique)
Faculté des Sciences et des Techniques
Université de Rouen
76128 Mont Saint-Aignan Cedex, France

e-mail: {Christian.Charras,Thierry.Lecroq}@dir.univ-rouen.fr

Abstract. We present an animation in Java for exact string matching algorithms [4]. This system provides a framework to animate in a very straightforward way any string matching algorithm which uses characters comparisons. Already 27 string matching algorithms have been animated with this system. It is a good tool to understand all these algorithms.

Key words: Exact string matching, animation, Java

1 Introduction

Pattern matching is a very important field in computer science as much from a theoretical viewpoint as from a practical one. It occurs for instance in text processing, speech recognition, information retrieval, and computational biology. It also provides challenging theoretical problems. For a large number of programs, the techniques used to match a pattern constitute a high percentage of their total work. It is then important to design very efficient algorithms. Understanding the existing algorithms is helpful to achieve this goal. It seemed to us very interesting to offer a tool to visualize easily string matching algorithms.

String matching is a special case of pattern matching where the pattern is set up by a finite sequence of characters. It consists in finding one, or more generally, all the occurrences of a word x of length m in a text y of length n. Both x and y are built over the same alphabet Σ.

The best way to understand how a string matching algorithm works is to imagine that there is a window on the text. This window has the same length as the word x. It is first aligned with the left end of the text y, then the string matching algorithm scans if the symbols of the window match the symbols of the word (this specific work is called an *attempt*). After each attempt, the window (and the word) is shifted to the right over the text until it goes beyond the end of the text. A string matching algorithm is then a succession of attempts and shifts. The aim of an efficient algorithm is to minimize the work done during each attempt and to maximize the length of the

[1]This work was partially supported by the project "Informatique et Génomes" of the french CNRS.

shifts. To achieve this, most of the string matching algorithms preprocess the word before the searching phase. All the different string matchings algorithms differ both in the way they compute the attempts (from left to right, from right to left or from other specific orders) and in the way they compute the shifts.

Numerous solutions to the string matching problem have been designed (see [6] and [11]). The two most famous are the Knuth-Morris-Pratt algorithm [9] and the Boyer-Moore [2]. There exist then a large number of algorithms using various methods. It is interesting to have a tool to understand them. There exist some general-purpose visualization systems (see [10] and [3]). These systems have been developed for X Window. Such a system, running for X Window and dedicated to string matching, has been developed by Baeza-Yates and Fuentes [1]. Some specialized systems are accessible directly through the World Wide Web (see [7], [8], [12] and [13]). All of these systems enable only to visualize the very classical string matching algorithms and usually they do not permit to keep trace of the history of the search phase. Our system offers to the users the possibility to follow the running of a large choice of string matching algorithms very easily through the World Wide Web.

This article is organized as follows: Section 2 described how the system operates and Section 3 described how the system is written and how to animate a new string matching algorithm.

2 The environment

The user can choose among the 27 string matching algorithms already implemented. For each algorithms there is a button (see Figure 1). If one clicks on a button, a window appears (see Figure 2). In that window the user can then enter a text and a word (default text is `gcatcgcagagagtatacagtacg` and default word is `gcagagag`). The text and the word alphabet is restricted to the lower case letters. A button enables then the user to start the search and another button to stop it at any time. The search phase is then shown attempt by attempt: for one attempt the text is displayed and the word, which all characters are materialized by a dot, is aligned with the relevant position in the text. In each attempt the different character comparisons are shown in the following way:

- matches are shown by displaying the word letter in upper case;
- mismatches are shown by displaying the word letter in lower case.

An occurrence of the word in the text is shown by displaying the corresponding text characters in upper case. At the end of the search phase, the system gives the number of attempts and the number of character comparisons performed during the search phase (see Figure 2).

3 The model

The system is written in Java. It is dedicated mainly to exact string matching algorithms but is easily extensible to a large family of algorithms. Moreover it is completely straightforward to implement any string matching algorithms providing that it is written in a specific way.

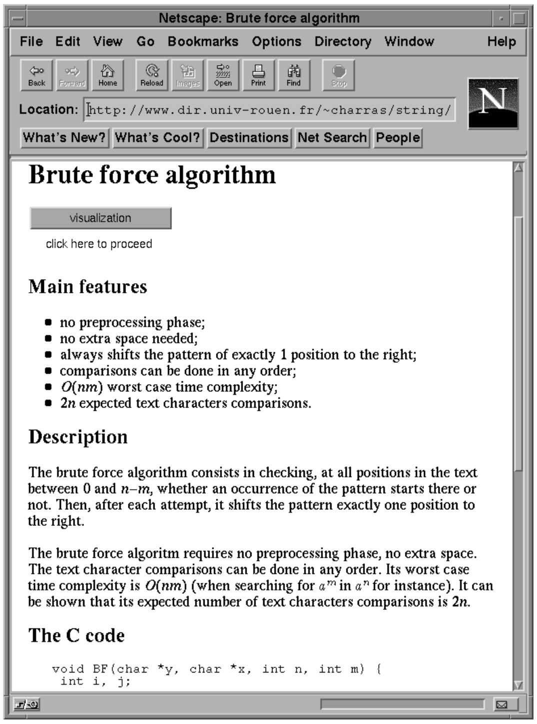


Figure 1: The visualization button for the Brute Force algorithm.

Let us first describe the different structures used by the system. The different buttons for each string matching algorithms are dealt by a class called `AppletButton2` which inheritance graph is shown Figure 3 (`AppletButtoni` is an applet with i inputs for $0 \leq i \leq 3$).

The windows displaying the search phases are dealt by a class which name is `ProgramTextWindow2` which inheritance graph is shown Figure 4.

All the different string matching algorithms inherit of a class which name is `ProgramSPM` which inheritance graph is given Figure 5.

In `ProgramSPM` the different following methods are declared:

- `showAttemptAt(i)`: this method displays m dots below the position i in the text;
- `EqCharsAt(i,j)`: this method tests, and displays the word character accordingly, if there is a match between characters y_i and x_j;
- `NotEqCharsAt(i,j)`: this method tests, and displays the word character accordingly, if there is a mismatch between characters y_i and x_j;
- `showMatch(i)`: this method displays a full match of the word at position i in the text;
- `showComparisons()`: this method displays the number of attempts and the number of character comparisons at the end of the search phase.

The word x, its length m, the text y and its length n are all attributes of the class `programSPM`.

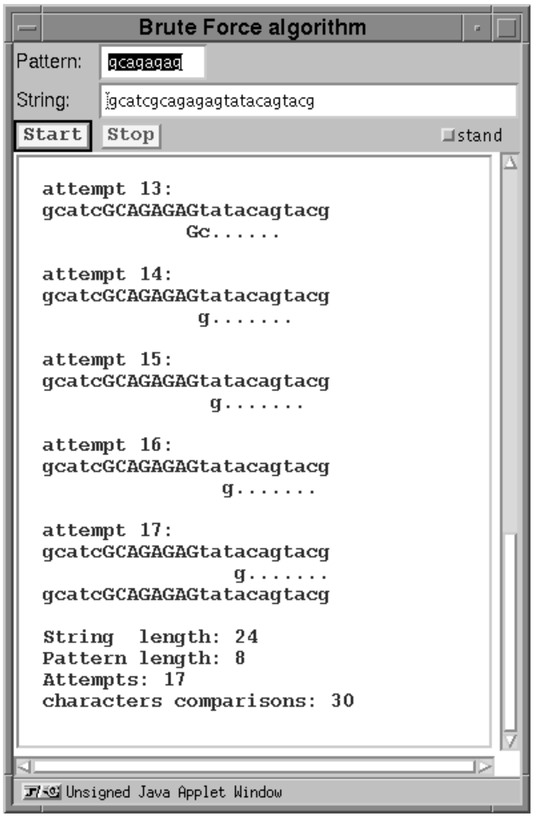


Figure 2: The window for the Brute Force algorithm after a run.

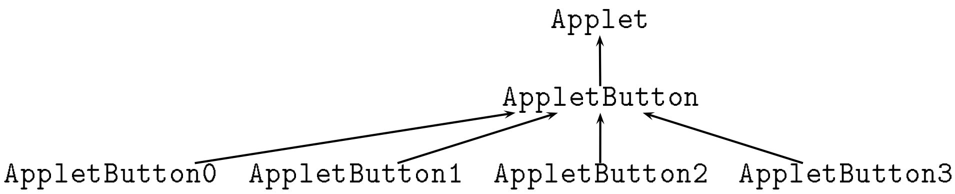


Figure 3: Inheritance graph of the button class.

The animation of a string matching algorithm is very easy if the begin of each attempt (`showAttemptAt`), the character comparisons (`EqCharsAt` or `NotEqCharsAt`) and the report of a full occurrence (`showMatch`) are clearly identified and separated from any other instruction.

Thus the translation of the Brute Force string matching algorithm (see Figure 6) is very straightforward (see Figure 7).

And for a more complicated algorithm as for the Colussi algorithm [5] it is as simple (see Figure 8 and 9).

4 Concluding Remarks

We have presented a system which is able to animate exact string matching algorithms. A demo package is available at the following address:

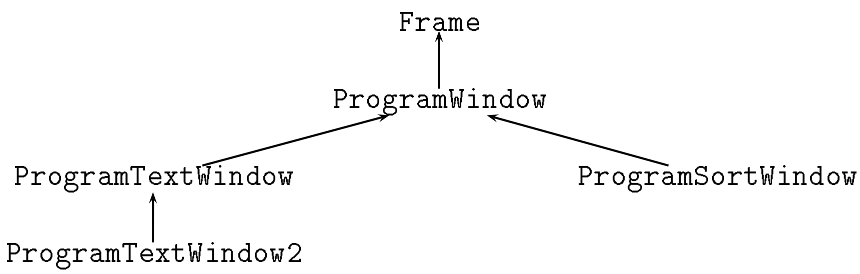


Figure 4: Inheritance graph of the window class.

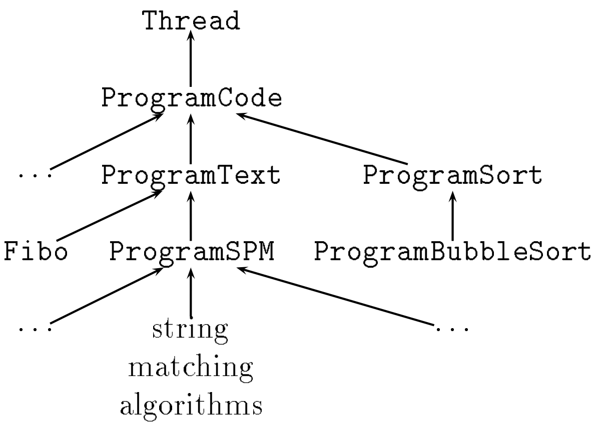


Figure 5: Graph inheritance for the string matching algorithms.

```
void BF(char *y, char *x, int n, int m) {
   int i, j;

   for (i=0; i <= n-m; i++) {
      j=0;
      while (j < m && y[i+j] == x[j]) j++;
      if (j >= m) OUTPUT(i);
   }
  }
```

Figure 6: Brute Force string matching algorithm in C.

```
import lirdir.aptk.InterruptionException;
import lirdir.progtext.ProgramSPM;

public final class ProgramBruteForce extends ProgramSPM {

    public void MAIN() throws InterruptionException{
        int i, j;
        for (i=0; i <= n-m; i++) {
            showAttemptAt(i);
            j = 0;
            while (j < m && EqCharsAt(i+j,j)) j++;
           if (j >= m) showMatch(i);
        }
        showComparisons();
    }
}
```

Figure 7: Brute Force string matching algorithm in Java.

```
void COLUSSI(char *y, char *x, int n, int m)
{
 int i, j, right, last, nd, h[XSIZE], next[XSIZE], shift[XSIZE];

 PRE_COLUSSI(x, m, h, next, shift, &nd);

 /* Searching */
 i=0;
 right=0;
 last=-1;
 while (i <= n-m) {
    j=right;
    while (j < m && last < i+h[j] && y[i+h[j]] == x[h[j]]) j++;
    if (j >= m || last >= i+h[j]) {
       OUTPUT(i);
       j=m;
    }
    if (j > nd) last=i+m-1;
    i+=shift[j];
    right=next[j];
 }
}
```

Figure 8: Colussi string matching algorithm in C.

```
import lirdir.aptk.InterruptionException;
import lirdir.progtext.ProgramSPM;

public final class ProgramColussi extends ProgramSPM {

    public void MAIN() throws InterruptionException {
      int i, j, right, last, nd;
      int h[] = new int[m+1];
      int next[] = new int[m+1];
      int shift[] = new int[m+1];
      nd = PRE_COLUSSI(h, next, shift);
      /* Searching */
      i=0;
      right=0;
      last=-1;
      while (i <= n-m) {
          showAttemptAt(i);
          j=right;
          while (j < m && last < i+h[j] && EqCharsAt(i+h[j],h[j])) j++;
          if (j >= m || last >= i+h[j]) {
             showMatch(i);
             j=m;
          }
          if (j > nd) last=i+m-1;
          i+=shift[j];
          right=next[j];
      }
      showComparisons();
    }
}
```

Figure 9: Colussi string matching algorithm in Java.

`ftp.dir.univ-rouen.fr/pub/ESMAJ/esmaj.zip`
and can be consulted at
`http://www.dir.univ-rouen.fr/~charras/esmaj/`.

We have shown how it is easily possible to animate new string matching algorithms providing that they are written in a given form. This system can easily be extended to animate other class of algorithms. It seems quite obvious that animating approximate string matching algorithms would just need a few efforts. Some sort algorithms and some graph algorithms have already been animated with the same principles.

References

[1] R.A. Baeza-Yates and L.O. Fuentes. A framework to animate string algorithms. *Inform. Process. Lett.*, 59(5):241–244, 1996.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.

[3] M.H. Brown. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[4] C. Charras and T. Lecroq. Exact string matching algorithms, 1996. URL:`http://www.dir.univ-rouen.fr/~charras/string/`

[5] L. Colussi. Correctness and efficiency of the pattern matching algorithms. *Inform. Comput.*, 95(2):225–251, 1991.

[6] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[7] A. Cássia Rossi de Almeida. Smaa: string matching algorithm animation. URL:`http://www.dcc.ufmg.br/~cassia/english_version_smaa.html`

[8] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, 1998. URL:`http://www.cgc.cs.jhu.edu/~goodrich/dsa/11strings/demos/pattern/`

[9] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[10] J. T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Trans. Comput.*, 23(9):27–39, 1990.

[11] G. A. Stephen. *String searching algorithms.* World Scientific Press, 1994.

[12] M. Takeda. Demonstration of naive, KMP, and BM pattern matching algorithms, and their variations. URL:`http://www.i.kyushu-u.ac.jp/~takeda/PM_DEMO/e.html`

[13] K. A. Zaman. Illustrated pattern matching. URL:`http://www.cs.columbia.edu/~zkazi/proj.html`

Local Prediction for Lossless Image Compression

Ahmad Daaboul

Institut Gaspard Monge
University of Marne-la-Vallée
Cité Descartes, 5, Bd Descartes, Champs-sur-Marne
77454 Marne-la-Vallée CEDEX 2
France

e-mail: `daaboul@monge.univ-mlv.fr`

Abstract. In predictive coding a group of neighboring picture elements is used to select a suitable prediction value for a current pixel. In this paper, we propose two techniques for lossless images compression based on predictive coding. In the first technique which called, *the predictors*, we replace each pixel in the image by the predicted pixel; we use various schemes to predict the value of a pixel. In the second, which is based on predictor technique, and called *optimal prediction schemes*, we divide the original image into blocks or lines and seek the best predictor for each (among a selected set of eight) that provides the best prediction. The errors image is encoded through arithmetic coding, during the final step of compression. The gains of compression that we obtained are observed in the lossless image compression.

Key words: Image compression and Predictive coding.

1 Introduction

The aim of image compression is to represent a given image with the minimal number of bits in order to accelerate transmission or reduce storage. Image compression can be divided into two categories. In the first, *lossy compression* : we accept a difference between the original image and the decompressed one. Second, *lossless compression*: after a cycle of compression/decompression, the decompressed image is identical to the original image.

Most image compression techniques are lossy. However, there are many applications which require lossless compression. For example in medical and satellite images no loss of information can be tolerated. In our work we focused lossless image compression.

Among the various methods which have been devised for lossless compression, *predictive coding* is perhaps the most simple and efficient. In predictive coding, a prediction is made for the current pixel based on the values of previously encountered neighboring pixels. For every input x_N pixel, a *predictor* generates a prediction value which is calculated from $N-1$ preceding samples. A *predictor* is a linear or non-linear combination of neighboring pixels of a current pixel. We call *error image* the difference between the original image and the predicted image. If the prediction scheme is satisfactory then the distribution of prediction error is concentrated near zero and the error image has a significantly lower entropy compared to the original image. Lossless image compression techniques [TLR85] identify two basic steps: *decorrelation* and *coding*. In the decorrelation step, redundancies among the pixels are reduced. In the coding step, the *error image* is encoded into a binary string using a variable length code, such as a *Huffman coding* [Hu52] or *arithmetic coding* [BCW90, R76].

The predictors proposed by Wallace [W91] define the JPEG lossless image compression standard. Harrison [H52] proposed two predictors, others were defined by Todd, Langdon and Rissanen [TLR85]. In this paper, we describe new predictors. Secondly, we select some JPEG predictors, Harrison predictors and two of our predictors. This set of selected predictors is used to predict a sample of pixels and we choose (among the set of selected predictors) the one that provides the best prediction for this sample. The error image obtained is encoded by a zero order arithmetic coding.

We consider an image to be an array P of integers of two dimensions $M \times N$ such that $0 \leq m < M$ and $0 \leq n < N$, where M denotes the number of lines of P and N, the number of columns. In this paper Pr represents a predictor.

2 Predictors techniques

Among all the methods of lossless image compression, the methods based on predictors are the simplest. These methods take into account the value of a pixel compared to its neighbors. Different predictors have been proposed to predict the value of a pixel at the location (m, n). Harrison [H52] has proposed some predictors. These predictors are called *slope predictor* (Pr_s):

$$Pr_s(m,n) \ = 2 * P[m, n-1] - P[m, n-2].$$

and *Plane 3 predictor* (Pr_{p3}):

$$Pr_{p3}(m,n) \ = \frac{2}{3}P[m, n-1] + \frac{2}{3}P[m-1, n] - \frac{1}{3}P[m-1, n-1].$$

Other predictors are defined in [TLR85], they are called *Plane* 2 *predictor* (Pr_{p2}):

$$Pr_{p2}(m,n) \ = P[m, n-1] + (P[m-1, n+1] - P[m-1, n-1])/2.$$

and *Right diagonal* (Pr_{p3}):

$$Pr_{rd}(m,n) \ = P[m-1, n-1].$$

Table 1 contains the predictors proposed by Wallace [W91] which are used as the JPEG lossless image compression standard.

Predictor	Prediction
$Pr_{J1}(m,n)$	$P[m-1,n]$
$Pr_{J2}(m,n)$	$P[m,n-1]$
$Pr_{J3}(m,n)$	$P[m-1,n-1]$
$Pr_{J4}(m,n)$	$P[m-1,n]+P[m,n-1]-P[m-1,n-1]$
$Pr_{J5}(m,n)$	$P[m-1,n]+((P[m,n-1]-P[m-1,n-1])/2)$
$Pr_{J6}(m,n)$	$P[m,n-1]+((P[m-1,n]-P[m-1,n-1])/2)$
$Pr_{J7}(m,n)$	$(P[m-1,n]+P[m,n-1])/2$

Table 1: JPEG predictors.

In this work, we propose new predictors which correspond to different schemes of linear prediction. Table 2 contains our predictors.

Predictor	Prediction
$Pr_1(m,n)$	$(2*P[m,n-1]+P[m-1,n-1])/3$
$Pr_2(m,n)$	$(2*P[m,n-1]+P[m-1,n])/3$
$Pr_3(m,n)$	$max(P[m,n-1],P[m-1,n])$
$Pr_4(m,n)$	$(P[m,n-1]+P[m-1,n]+P[m-1,n-1])/3$
$Pr_5(m,n)$	$(3*P[m,n-1]+P[m-1,n]+P[m-1,n-1])/5$
$Pr_6(m,n)$	$max(P[m,n-1],P[m-1,n],P[m-1,n-1])$
$Pr_7(m,n)$	$(P[m,n-1]+P[m-1,n]+P[m-1,n-1]$ $+P[m-1,n+1])/4$
$Pr_8(m,n)$	$(P[m,n-1]+P[m-1,n])+P[m-1,n-1]$ $+P[m-1,n+1]+P[m-1,n+2])/5$

Table 2: Our predictors.

3 Optimal prediction schemes

The best method to predict a pixel is to compare it with its neighbors in the same sample, and select the neighbor that provides the best prediction, i.e. the nearest value among his neighbor. Beginning with this idea, we search for the best predictor for a sample of pixels, not of only one. For this purpose, we divide the original image into *line* and *block* as described in sections 3.1 and 3.2. The basic idea of our technique is to use a sample of predictors to predict an image or a part of an image. This sample is composed by eight predictors : Pr_{J1}, Pr_{J2}, Pr_{J5}, Pr_{J6}, Pr_{J7}, Pr_{p3}, Pr_2 and Pr_3. The reason for choosing these eight predictors is that the values of these predictors are used to detect the magnitude and orientation of edges in the input image (or sample of pixels) and make necessary adjustments in the prediction.

We predict the sample pixels using all predictors, and we compute the *zero-order entropy* of error image with each of them. The best predictor is the one that provides the lower zero-order entropy, which is given by the formula defined below.

If we have n independent symbols whose probabilities of choice are P_i, then we define the *zero-order entropy* as follows:

$$E = - \sum_{0 \prec i \preceq n} P_i \log(P_i)$$

3.1 Lines partitioning schemes

Here the image will be divided into lines, each line is of size 256. Afterwards, every line will be divided into a *Vector-Line* type. Denote, V-L$_i$[x] a Vector-Line of size x, the V-L$_i$ types are:

$$\text{V-L}_i[2^i], \text{ such that } 1 \leq i \leq 8.$$

Figure 1 shows the main steps using lines partitioning schemes:

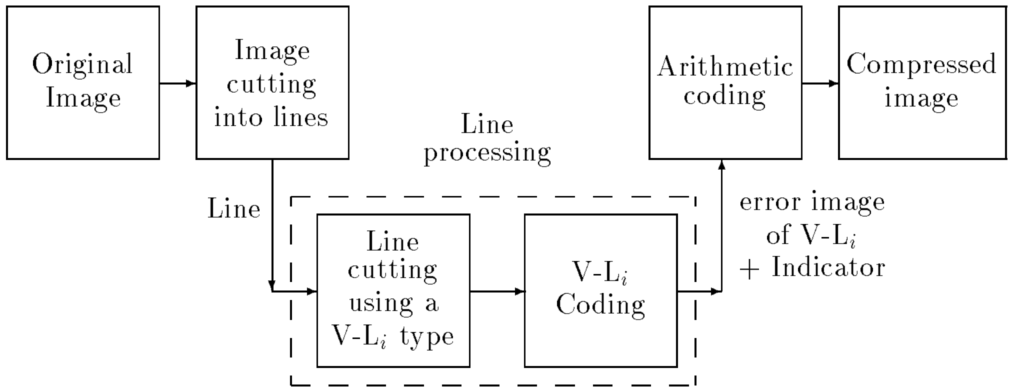


Figure 1: Lines partitioning schemes

3.1.1 V-L Encoding

Each line of the original image, is divided using a type of V-L$_i$. For example if i=7, V-L$_7$, we divide a line of the original image into two vectors of size 128 each. Let V-L$_7$[128] a vector of V-L$_7$ type, find below the main steps to encode this vector:

1. processing of V-L$_7$[128] using the eight predictors ;
2. calculation of V-L$_7$[128] error image and of the zero-order entropy of error image for each predictor ;
3. selection of the best among the eight predictors, the one which provides the lower zero-order entropy of V-L$_7$[128] error image ;
4. the V-L$_7$[128] error image , calculated using the selected predictor, and the indicator[1] which is used to indicate the selected predictor are processed with zero-order arithmetic coding [BCW90, R76].

[1]The indicator is an integer from 0 to 7, of the eight predictors.

To provide the best way of searching for the best predictor, we need to process the image in steps. First, we process the image to detect the magnitude and the orientation of edges in the input image. Secondly, according to the output of the first step, we choose the best predictor and the best block size.

3.2 Blocks partitioning schemes

Here, we divide the image into a *Vector-Block[M][N]* type where M is the number of lines, and N is the number of columns. Next, we process every Vector-Block by the eight predictors and select the predictor that provides the lower zero-order entropy. We use *V-B* to denote a Vector-Block. We describe the V-B types and their length as follows:

$$\begin{cases} \text{V-B}_i[2^i][2^i] & \text{such that } 2 \leq i \leq 7 \\ \text{V-B}_i[2^{i-1}][2^i] & \text{such that } i = 8. \end{cases}$$

To encode a V-B, we follow the same steps as the V-L encoding.

4 Results

We tested the presented algorithms on eight images. All the images are of size 256×256 and have 256 intensity levels. The images have been extracted from the university of Southern California and Nebraska-Lincoln Database. These images are part of a standard test set used by the image compression research community. The gain of compression is computed in the following way:

$$\%\ \text{gain} = \frac{R_o - R_e}{R_o} \times 100$$

where R_o is the size of original image and R_e is the size of compressed image. Table 3 contains the gains of compression using the following predictors: Pr_{rd} and Pr_s, Pr_{p3}, Pr_{p2} and trivial predictor (the original image).

Image	$Trivial$	Pr_{rd}	Pr_s	Pr_{p2}	Pr_{p3}
USG-Girl	21.0%	31.6%	29.6%	36.0%	38.2%
Girl	21.7%	41.0%	32.6%	39.1%	39.2%
Lady	36.3%	45.6%	42.2%	49.1%	49.9%
House	21.8%	33.8%	35.5%	40.7%	42.7%
USC-Couple	26.7%	36.1%	34.3%	41.6%	45.1
Tree	10.6%	22.8%	22.5%	27.9%	30.4%
Satellite	09.1%	15.1%	17.0%	22.0%	25.3%
X-Ray	**34.7%**	31.2%	15.9%	17.9%	19.0%

Table 3: Gain obtained using trivial and Harrison predictors.

Image	$J1$	$J2$	$J3$	$J4$	$J5$	$J6$	$J7$
USG-Girl	35.4%	35.9%	31.7%	34.5%	37.2%	36.8%	**38.7**%
Girl	44.8%	**46.1**%	41.3%	39.5%	40.1%	39.3%	42.5%
Lady	47.6%	**51.3**%	45.7%	46.8%	50.4%	48.2%	**51.2**%
House	37.2%	41.9%	33.6%	41.4%	**43.4**%	41.5%	42.1%
USC-Couple	43.1%	41.3%	35.7%	43.5%	44.2%	**45.3**%	44.2%
Tree	25.4%	29.8%	24.0%	26.8%	30.4%	28.5%	**30.9**%
Satellite	19.4%	22.3%	17.5%	21.8%	24.7%	23.5%	**25.4**%
X-Ray	**34.5**%	34.3%	31.2%	24.6%	20.0%	20.1%	23.3%

Table 4: Gain obtained using JPEG predictors.

Image	Pr_1	Pr_2	Pr_3	Pr_4	Pr_5	Pr_6	Pr_7	Pr_8
USG-G	35.4%	**38.3**%	37.5%	36.7%	37.1%	36.3%	37.8%	37.0%
Girl	39.6%	41.3%	**47.8**%	39.7%	40.1%	47.0%	39.9%	39.4%
Lady	50.4%	**51.6**%	50.7%	50.4%	51.2%	50.4%	50.7%	50.1%
House	40.7%	**42.9**%	41.2%	39.6%	41.6%	40.0%	39.7%	38.6%
USC-C	40.3%	**43.7**%	43.5%	41.4%	42.0%	42.0%	42.4%	40.9%
Tree	29.7%	**31.6**%	28.6%	29.1%	30.7%	27.4%	28.6%	27.4%
Satellite	22.8%	**25.6**%	22.9%	23.2%	24.4%	21.6%	22.9%	21.1%
X-Ray	20.0%	21.3%	**36.7**%	20.6%	20.3%	36.0%	20.1%	20.0%

Table 5: Gains obtained using our predictors.

Image	V-L_1	V-L_2	V-L_3	V-L_4	V-L_5	V-L_6	V-L_7	V-L_8
USG-G	27.8%	29.5%	33.5%	36.1%	37.3%	37.8%	37.9%	**38.1**%
Girl	39.3%	40.5%	43.3%	45.7%	47.6%	**48.3**%	**48.3**%	48.0%
Lady	41.1%	42.6%	46.3%	49.2%	51.3%	52.2%	52.4%	**52.6**%
House	31.7%	33.7%	38.3%	41.2%	43.0%	43.9%	43.9%	**44.1**%
USC-C	33.4%	37.2%	41.9%	44.5%	45.8%	**46.0**%	45.8%	45.7%
Tree	22.4%	22.5%	25.7%	28.6%	30.4%	**31.2**%	**31.2**%	31.1%
Satellite	15.7%	15.9%	19.3%	22.1%	23.9%	25.0%	25.5%	**25.7**%
X-Ray	26.9%	25.6%	28.9%	32.8%	35.7%	36.3%	36.4%	**36.5**%

Table 6: Gains obtained using our lines partitioning schemes.

Image	V-B$_1$	V-B$_2$	V-B$_3$	V-B$_4$	V-B$_5$	V-B$_6$	V-B$_7$	V-B$_8$
USG-G	18.4%	30.2%	36.1%	37.8%	38.0%	38.5%	38.6%	**38.7**%
Girl	27.0%	39.9%	46.0%	47.8%	**47.9**%	47.8%	47.8%	47.8%
Lady	30.9%	43.8%	51.1%	53.0%	**53.2**%	52.4%	50.9%	51.6%
House	20.1%	34.2%	42.1%	44.7%	**45.2**%	44.1%	42.8%	43.4%
USC-C	27.9%	39.1%	44.9%	**46.4**%	46.2%	46.3%	45.9%	46.0%
Tree	11.0%	22.5%	29.2%	31.1%	31.3%	31.0%	31.2%	**31.6**%
Satellite	03.9%	15.4%	22.1%	24.7%	25.3%	25.5%	**25.8**%	**25.8**%
X-Ray	21.9%	27.9%	34.6%	36.0%	36.3%	36.6%	36.6%	**36.7**%

Table 7: Gains obtained using our blocks partitioning schemes.

Tables 3, 4 and 5 contain the gains of compression using JPEG, Harrison and our predictors. If we compare these tables we notice that the gains in table 5 are often higher than those in tables 3 and 4. This means that the predictions that we propose are often better than those proposed by Harrison and Wallace. Table 6 and 7 contain the gains of compression using lines partitioning schemes and blocks partitioning schemes that we propose. The results in the two tables are clearly better than those in table 3, 4 and 5.

Conclusions

In this paper, we presented two techniques of lossless image compression based on predictors. If we compare our performances with the existing algorithms based on the refinement of pixels and specialized in lossless image compression, we obtain higher results. The algorithms that perform the best, are Pr_2 and Pr_3, which provide the best prediction. The optimal prediction scheme techniques obtains the best results.

References

[BCW90] Bell, T.C., Cleary, J.C. and Witten, I.H.: Text Compression. Advanced Reference Series. Prentice-Hall, Englewood Cilffs, New Jersey, 1990.

[H52] Harrison, C.W.: Experiments with linear prediction in television. Bell System Tech. J., 31, pages 764–783, July, 1952.

[Hu52] Huffman, D.A.: A method for the construction on minimum redundancy codes. Proc. IRE, 40, pages 1098–1101, 1952.

[R76] Rissanen, J.J.: Kraft inequality and Arithmetic coding. IBM Journal of Research and Development, 20, pages 198–203, May, 1976.

[TLR85] Todd, S., Langdon, G.G. and Rissanen, J.: Parameter reduction and context selection for compression of gray scale images. IBM J. Res. Develop., 29, 2, pages 188–193, March 1985.

[W91] Wallace, G.K.: The JPEG still picture compression standard. Communications of the ACM, 34, 4, pages 31–44, 1991.

On the All Occurrences of a Word in a Text

O.C. Dogaru

West University of Timişoara
Bd.V.Pârvan,nr.4,Timişoara,1900,Romania

e-mail: `dogaru@info.uvt.ro`

Abstract. In this paper a simple straight string search algorithm is presented. For a string s that consists of n characters and a pattern p that consists of m characters the order of comparisons is $O(n.m)$, $0 < m \leq n$, in the worst case, but the average time complexity is good. The algorithm presented finds all occurrences of p in s. It do not use a precompiling of the pattern p.

1991 Mathematical Subject Classifications: 68P10 [Searching and Sorting]

Key words: direct, string, pattern, search

1 Introduction

The string matching problem is following. Given an array $s[0..n-1]$ of n characters and an array $p[0..m-1]$ of m characters where $0 < m \leq n$, the task is to find all occurrences of p in s. The string s is regarded as a *text* and the string p as a *word*(*pattern*). Generally, s and p are item.

In [W86] it is presented a direct method to determine the first occurrence of p in s. In the same book it is presented the fact that the algorithm proposed is very inefficient, for example, if the pattern is p=$a^{m-1}b$ and the string is s=$a^{n-1}b$, then $m * n$ comparisons are necessary to determine that p is in s.

In this direct method the pattern and the text are aligned at the left ends. The searching begins with p_0 and s_0. If a mismatch appears then a new searching begins always with p_0, the first character of the pattern.

2 The algorithm

The algorithm proposed by us begins with p and s aligned at the left ends too but in the case that a mismatch occurs in the process of comparisons of p and s ($p_j \neq s_j$) then the searching continues with the character of p which produced the mismatch, that is p_j, which is searched between s_{j+1} and s_{n-m+j}. On this idea the algorithm is built. It will contain the followings.

1. One compares successively p_0 with s_i, i=0,1,...,$n-m$. If it exists no match of the p_0 with s_i, i=0,1,...,$n-m$ then 'p is not in s' and the process is terminated.

2. If s_i is the first match of p_0 then one compares successively p_1 with s_{i+1}, p_2 with s_{i+2} etc. If all p_j match with s_{i+j}, j=0,1,...,m-1 then this is the first occurrence of p in s. A new searching is resumed beginning with p_0 and s_{i+m}.

3. If in the process of searching a mismatch occurs between p_j and s_{i+j} ($p_j \neq s_{i+j}$) then p_j is searched in the rest of string s between s_{i+j+1} and s_{n-m+j}. If p_j is not in this rest then the searching is ended.

4. If in the substring $s_{i+j+1}, ..., s_{n-m+j}$ there exists a character which match with p_j, one renames this character s_i. Therefore $p_j = s_i$. In this case one compares the left and right neighbours of p_j and s_i that is $p_0, p_1, ..., p_j, ...,$
p_{m-1} with correspondings $s_{i-j}, ..., s_i, ..., s_{i-j+m-1}$. If all occur then this is an occurrence of p in s and the process of searching is resumed. If in the time of verification the neighbours of p_j and s_i a mismatch occurs then a new searching of p_j begins with the character s_{i+1}.

5. The algorithm stops if $i >= n - m + j$.

Example.

```
p=abcd  (m=4)
s=xabcdxabxxaycdxabcd    (n=19)
  a
   abcd
       a
        abc
           c
            c
             c
              c
            a?c
               c
                c
                 c
                  c
                 abcd
```

In this example there are 23 comparisons to find two occurrences of p in s.

The complete algorithm, presented as a procedure named DO3(written in a Pascal-like language described in [HS83]), is the following.

```
  procedure DO3(s,p,n,m)
 //find all occurrences of the word p(0:m-1)//
 //in the string s(0:n-1) if this exists. If yes//
         //then procedure writes 'p is in s' else it//
 // write 'p is not in s'. 0<m<=n//
      char p(0:m-1),s(0:n-1); integer i,j,m,n,k; boolean f;
      i:=0; f:=false;
      loop
        j:=0;
        while (j<m) and (p(j)=s(i)) do i:=i+1;j:=j+1 repeat;
        if (j=m) then write('p is in s');f:=true;cycle endif
          // the character p(j) is a mismatch:p(j)<>s(j) //
        1:i:=i+1;
        while (i<=n-m+j)and(p(j)<>s(i)) do i:=i+1 repeat
```

```
    if i>n-m+j and not f then exit endif;
      // it exists i thus p(j)=s(i),one verifies the //
      //left and right neighbours of p(j) and s(i)//
    k:=0;
    while(k<=m-1) and (p(k)=s(i-j+k) do k:=k+1 repeat;
    if k=m then write('p is in s'); f:=true; i:=i-j+m
           else  goto 1  endif
  until i>=n-m+j repeat;
  if not f then write('p is not in s') endif
endDO3;
```

3 Number of comparisons

The maximum number of comparisons to determine that 'p is or it is not in s', theoretically, it is obtained when, after $p_k = s_k, k = 0, 1, ..., j-1$ match, it appears $p_j \neq s_j$, but $p_j = s_i, i = j+1, ..., n-m+j$ and all the left neighbours of p_j match with the corresponding neighbours of s_i and the right neighbours of p_j, that is, $p_{j+1}, p_{j+2}, ..., p_{m-2}$ match with the right corresponding neighbours of s_i excepting p_{m-1}. For $i = n-m+j, p_{m-1}$ may or it may not match with his corresponding in s. Therefore for:

$i = j+1$, $p_0 = s_1, ..., p_j = s_i, ..., p_{m-2} = s_{m-1}; p_{m-1} \neq s_m$ there are m comparisons;

$i = j+2$, $p_0 = s_2, ..., p_j = s_i, ..., p_{m-2} = s_m; p_{m-1} \neq s_{m+1}$ there are m comparisons;

....

$i = n-m+j$, $p_0 = s_{n-m+j}, ..., p_j = s_i, ..., p_{m-2} = s_{n-2}$ and $p_{m-1} = s_{n-1}$ or $p_{m-1} \neq s_{n-1}$, there are m comparisons. Therefore in all it exists $j+1$ comparisons p_k with $s_k, k = 0, 1, ..., j$; between $j+1$ and $n-m+j$ there exists $(n-m+j)$-$(j+1)+1 = n-m$ cases for which p_j may match with $s_i, i = j+1, j+2, ..., n-m+j$ and the neighbours of p_j, that is $p_0, p_1, ..., p_{m-2}$ match with the corresponding neighbours of s_i, but $p_{m-1} \neq s_{m+k}, k = -1, 0, 1, ..., n-m-1$. Possibly, $p_{m-1} = s_{n-1}$. Every case gives m comparisons. Hence the maximum number of comparisons is

$$N_{max} = j + 1 + (n-m) * m \leq m - 1 + 1 + (n-m)m = m(n-m+1).$$

The complexity of the algorithm DO3 is $\mathcal{O}(n.m)$ too.

But in the most unfavourable cases the algorithm DO3 reduces the maximum number of comparisons from $m * n$ as in algorithm presented by N.Wirth in [W86] to $m(n-m+1)$.

For the example $p{=}a^{m-1}b$ and $s{=}a^{n-1}b$ presented in Section 1, the algorithm DO3 carries out $n+m-1$ comparisons.

4 Profiling

The variant of this algorithm(OD) written to determine the first occurrence of p in s [D98] has been compared with a direct method(DIR) presented in [W86] and the Boyer-Moore algorithm(BM) [BM77].The tests have been realized for different values of $p(m$=5, 10, 20, 50, 100) and $s(n$=1000, 2000, 3000, 4000, 5000). The p and s have been generated randomly. One generated sequences of m and n decimal integer random numbers between 32-127 and one has tacken the ASCII corresponding characters for p respectively for s. For the same m and n the three methods have been executed 10 times. The average time for an m and five values for n(=1000, 2000, 3000, 4000, 5000) are written down in the following table

m=	5	10	20	50	100	Average
OD	0.52	0.20	0.56	0.24	0.30	0.364
DIR	0.46	0.58	0.24	0.34	0.58	0.432
BM	0.12	0.22	0.44	0.42	0.22	0.284

Between the average times of three methods there are the relations

$$t_{OD} = 1.28 t_{BM}; \quad t_{DIR} = 1.18 t_{OD}.$$

But if the three methods are executed 100 times then the values are the following

m=	5	10	20	50	100	Average
OD	0.362	0.374	0.396	0.376	0.368	0.372
DIR	0.408	0.398	0.408	0.414	0.396	0.404
BM	0.364	0.350	0.308	0.300	0.312	0.326

In this case the relations are

$$t_{OD} = 1.14 t_{BM}; \quad t_{DIR} = 1.08 t_{OD}.$$

5 Correctness of the algorithm

Theorem. *The algorithm DO3 works correctly.*
Proof. To proof the correctness of the algorithm we use a proof table [TBCG92]

procedure DO3(s,p,n,m)
char p(0:m-1),s(0:n-1); integer i,j,m,n,k; boolean f;
{pre:input=$(p_0, p_1, ..., p_{m-1}) \wedge (s_0, s_1, ..., s_{n-1}) \wedge$
$n \geq m > 0 \wedge \forall$i$\in${0,1,...,n-1}:$s_i$ are characters$\wedge$
$\forall$j$\in${0,1,...,m-1}:p_j are characters}
f:=false; i:=0;
loop
j:=0;

while (j<m) and (p(j)=s(i)) do
{inv:$\forall$ h$\in${0,1,...,j-1}:$p_h = s_h \wedge 0 \leq$j,i$\leq$m}
i:=i+1;j:=j+1 repeat;
{$\forall$ h$\in${0,1,...,j-1}:$p_h = s_h \wedge$(j=m$\vee p_j \neq s_i$}
if (j=m) then write('p is in s');f:=true;
{output f=true}
cycle endif
// the character p(j) is a mismatch:p(j)$\neq$ s(j) //
{f=false$\wedge$j<m$\wedge p_j \neq s_i$}
1:i:=i+1;
{0<i$\leq$n-m+j$\wedge p_j \neq s_i$)$\vee$i>n-m+j)}
while (i<=n-m+j)and(p(j)$\neq$s(i)) do
{inv:$p_j \neq s_{i-1} \wedge$ i$\leq$ n-m+j}
i:=i+1 repeat;
{($p_j \neq s_{i-1} \wedge$ $\neg$(i<=n-m+j$\wedge$ $p_j \neq s_i$)$\equiv$
{($p_j \neq s_{i-1}$ $\wedge$ i>n-m+j)$\vee$($s_i = p_j \wedge$ i<=n-m+j)}
if i>n-m+j and not f then
{$p_j \neq s_i$}
exit endif;
{$p_j = s_i \wedge$ i<=n-m+j}
// it exists i thus p(j)=s(i) //
//one verifies the left and right neighbours of p(j) and s(i)//
k:=0;
while(k<=m-1) and (p(k)=s(i-j+k) do
{inv:$\forall$h$\in${0,1,...,k-1}:$p_h = s_{i-j+h} \wedge$ 0$\leq k \leq$ m}
k:=k+1 repeat;
{($\forall$h$\in${0,1,...,k-1}:$p_h = s_{i-j+h}$)$\wedge$ $\neg$(k$\leq$ m-1$\wedge$ $p_k = s_{i-j+k}$)}
$\equiv$($\forall$ k$\in${0,1,...,m-1}:$p_k = s_{i-j+k} \wedge$ k=m)$\vee$
($\forall$h$\in${0,1,...,k-1}:$p_h = s_{i-j+h} \wedge$ $p_k \neq s_{i-j+k}$})
if k=m then write('p is in s'); f:=true; i:=i-j+m
{$\forall$k$\in${0,1,...,m-1}:$p_k = s_{i-j+k}$}
else
{$\exists$k$\in${0,1,...,m-1}:$p_k \neq s_{i-j+k}$}
goto 1
endif
until i>=n-m+j repeat;
{f=false $\vee$ f=true $\wedge$ 0<=j<=m$\wedge$ i>n-m+j}
if not f then write('p is not in s') endif;
{post:output=$\emptyset$}
endDO3;

The justifications are based on the application of logical equivalences and the rules of inference to the sequence of Pascal statements. These are:

i)*the assignment rule of inference*
{P(e)} v:=e {P(v)}

ii)*the conditional rules of inference*
a){P $\wedge$ B} s {Q}   b){P $\wedge$ B } s1 {Q}

$$\frac{P \wedge \neg B \Rightarrow Q}{\{P\} \text{ if } B \text{ then } s \ \{Q\}} \qquad \frac{P \wedge \neg B\} \ s2 \ \{Q\}}{\{P\} \text{ if } B \text{ then } s1 \text{ else } s2 \ \{Q\}}$$

iii)*the loop rules of inference*

$$\text{a)}\ \frac{\{inv \wedge B\} \ s \ \{inv\}}{\{inv\} \text{ while } B \text{ do } s \ \{inv \wedge \neg B\}} \qquad \text{b)}\ \frac{\{inv \wedge B\} \ s \ \{inv\}}{\{inv\} \text{ repeat } s \text{ until } B \ \{inv \wedge B\}}$$

where P,Q denote propositions, B-Boolean expression, inv-the invariant of the loop and s, s1, s2 are statements.

Conclusions

1)Algorithm OD is faster than algorithm DIR in average time;

2)There are pairs of p and s where algorithms OD or DIR are faster than algorithm BM;

3)At limit, the average times of the three methods tend to approach;

4)Possibly, for other p and s, the relations between the average times of the three methods can be slight different.

References

[BM77] R.S. Boyer, J.S. Moore, *A fast string searching algorithm*, Comm. ACM, **20**, 10(1977), pp.762-772

[CH92] R.Cole, R.Hariharan, *Tighter bounds on the exact complexity of string matching*, Proc. 33rd IEEE Symp. on Foundations of Computer Science,(1992), pp.600-609

[C94] R.Cole, *Tight bounds on the complexity of the Boyer-Moore string matching algorithm*, SIAM J.Comput, **23**, 5(1994), pp.1075-1091

[CHPZ95] R.Cole, R.Hariharan, M.Paterson, U.Zwick, *Tighter lower bounds on the exact complexity of string matching*, SIAM J. Comput., **24**, 1(1995), pp.30-45

[CH97] R.Cole, R.Hariharan, *Tighter upper bounds on the exact complexity of string matching*, SIAM J.Comput., **26**, 3(1997), pp.803-856

[CCGJLPR94] M.Crochemore, A.Czumaj, L.Gasiniec, S.Jarominek, T.Lecroq, W.Plandowski, W.Ritter, *Speeding up two string-matching algorithms*, Algoritmica, 5(1994), pp.247-267

[D93] O.Dogaru, *Algorithm of straight string search*, Proceedings of the 9th ROmanian SYmposium on Computer Science (ROSYCS), University of Iasi,(1993), pp.172-177

[D98] O.Dogaru, *On the first occurence of a pattern in a text*, Proceedings of MOSIS'98(Modelling and Simulation of Systems), International Conference, Volume 2, pp.45-50, May 5-7, 1998, Sv.Hostyn-Bistrice pod Hostynem, Czech Republic

[GG93] Z.Galil, R.Giancarlo, *On the exact complexity of string matching:Upper bounds*, SIAM J. Comput., **3**(1993), pp.407-437

[HS83] E.Horowitz, S.Sahni, *Fundamentals of Computer Algorithm*, Computer Science Press(1983)

[KMP77] D.E.Knuth, J.H.Morris, V.R.Pratt,*Fast pattern matching in string*, SIAM J.Comput. **6**, 2(1977), pp.323-349

[TBCG92] A.B.Tucker, W.J.Bradley, R.D.Cupper, D.K.Garnick, *Fundamentals of Computing* I, McGraw-Hill,Inc, (1992)

[W86] N.Wirth, Algorithm and Data Structures, Prentice Hall, N.J.(1986)

A Highly Parallel Finite State Automaton Processor for Biological Pattern Matching

Glen Herrmannsfeldt

Department of Molecular Biotechnology
University of Washington
Box 357730
Seattle, WA 98195-7730
USA

e-mail: `gah@mbt.washington.edu`

Abstract. Finite State Automata are useful for string searching problems mostly because they are fast. For very large problems, a software implementation will not be fast enough. I describe here a parallel implementation of a hardware Deterministic Finite State Automaton processor. It can rapidly search a large database for approximately matching strings, as a filter for more detailed processing later. As the most important parts, large Random Access Memory chips, are continually getting cheaper, it should be possible and affordable to make large arrays of such processors.

Key words: finite automata, approximate string matching, high-speed searching, deterministic finite automata, massive parallelism

1 Finite State Automata for Biology

An important problem in contemporary molecular biology is sequence comparison. One would like to compare DNA or protein sequences against other DNA or protein sequences and find ones that are most similar. As the database of known DNA and protein sequences is growing exponentially, this problem is continually getting harder. It is useful to have a machine to rapidly compare a group of sequences against a large disk file of sequences and indicate which ones match most closely.

All the examples will be done using a DNA alphabet size of four. The DNA database is much larger than the protein database, and so the larger, faster, processors are needed here first. The processors should be designed to also handle the protein 20 character alphabet.

This paper describes the design of hardware implementations of Finite State Automata [HU79][W87] for processing biological sequences. In all cases described here, the Finite State Automata are Deterministic, though sometimes the acronym FSA will be used instead of DFSA or DFA.

2 The Problem with Insertions and Deletions

The preferred method for approximate sequence comparison is dynamic programming [NW70]. Preferred, that is, unless you are in a hurry. Heuristic methods, based on hashing or finite state automata are faster than dynamic programming, but are not as good at separating the biologically related sequences from statistically insignificant, but similar looking sequences.

While finding an exact match fast is relatively easy, biological problems usually don't work that way. Both natural genetic mutations and errors in sequencing can cause inserted, deleted or substituted bases. Dynamic programming is well suited to doing comparisons with a supplied similarity matrix, giving the score for aligning any database character with any query character, and with specified insertion and deletion penalty values.[G82] Applying a dynamic programming algorithm will give a score for each database sequence aligned with the query sequence, and one can select the highest scoring sequences. The limitation is that dynamic programming is slow.

A modern RISC superscalar processor can do over 100,000,000 mathematical operations (additions or comparisons) per second. Using a dynamic programming algorithm with 10 operations per matrix element, where the number of matrix elements needed is the product of the query length and database length, we can calculate over 10,000,000 matrix elements per second. With a 1,000,000,000 base DNA database, and typical 1000 base query, one search requires 100,000 seconds, or about one day. To compare the entire 10^9 base GenBank against itself would require evaluating 10^{18} matrix elements in 10^{11} seconds. This is over 3000 years, clearly not practical.

One way to speed up the calculation is with special purpose hardware. Systolic array processors work very well for dynamic programming algorithms, [LL85][CH91] and are certainly the way to go to for fast dynamic programming. With an array of 10^3 processors, each evaluating 10^7 cells per second, the GenBank comparison can be done in 10^8 seconds, or about three years. But a large fraction of the sequences being compared will have no relation to the query sequence. If one could do an even faster comparison, as a filter before running the dynamic programming algorithm, it might be possible to speed up the whole process. It should at least be possible to find the interesting results sooner.

3 Finite State Automata for Biological Searching

The BLAST program is the most popular Finite State Automata implementation for biology.[AG90][KA90][KA93] It is probably the most popular overall. BLAST can rapidly scan a database to find possible matches, and then spend more time trying to extend those matches. With the default parameters, and some luck with the choice of query, BLAST may be within a factor of two of the speed that the data can be read off the disk. BLAST is very good at finding close matches, though its performance falls of for less exact matches. There are adjustable parameters which can be used to tune the search strategy. With parameters that do a better job at finding less exact matches, BLAST will slow down.

4 A Hardware Implementation of a Finite State Automaton

Searching with a Deterministic Finite State Automaton is very simple. The new state is obtained through a lookup table from the current state and the incoming database character. In hardware, this is not much more than a large RAM array and a register. One could imagine a single printed circuit board with a large number of processor elements, each consisting of a few RAM chips and a simple programmable logic chip, each processor implementing a DFSA. With the currently popular 64 Megabit RAM chips, and more recently available 256 Megabit chips, a lookup table with 8 million entries, and running with a 10 MHz clock should be very feasible. If one or more query sequence can be compiled into an 8 million state FSA, then we could get hundreds of queries on a system at one time, and search at near the disk streaming transfer rate. That is the motivation for this paper.

If we assume a 10 million character per second streaming rate, that a 1000 character query can be compiled into a single FSA processor, with 100 processors our comparison rate is 10^{12} per second. We can complete the GenBank against itself comparison in 10^6 seconds, about twelve days. This would be fast enough to allow multiple passes, to adjust parameters, and select sequences to feed to a slower, dynamic programming processor.

5 Finite State Automaton Size

A critical parameter is the size of the FSA necessary. Finite State Automata are very good at finding an exact match, as this requires relatively few states. For biological problems, it is necessary to include some substitution data, and usually also some insertions and deletions in the comparison. This is the reason we need such large FSAs. With dynamic programming, the scoring and gap penalties are part of the algorithm, and are used to calculate the score. With an FSA, it is necessary to predetermine all the sequences we will match, including sequences with substitutions, insertions, or deletions. We need a balance between the ability to find less exact matches, and the size of the FSA needed.

If we take a typical query of 1000 characters, and we search for an exact match, we need a 1000 state FSA. (Like BLAST, we trigger on transitions and not states, reducing the required state memory.) If we want to find any 16 character substring, we have 985 substrings of 16 characters each, for 15760 states.

At this point, it will be assumed that the number of states is equal to the total number of characters in the query substrings. Fewer states will be required, because of degeneracy at the beginning of the tree. Each state requires one table entry for each character in the alphabet. For now, we calculate just the total query size, and assume this is close to the number of states needed.

If we want to find any of the 16 character substrings with one of three possible substitutions from the DNA alphabet, in any one of the 16 positions, have to search for the exact match plus 48 possible mismatched strings, for each of the 985 substrings. Thus $49 \times 985 = 48265$ strings of 16 characters each, requiring 772240 states in the FSA. If we add all the single insertion and single deletion strings to the list, we increase by nearly a factor of three, approaching two million states.

This could be implemented as a two million entry look-up table, with each entry large enough to address the entire table, and in addition, to indicate the required result information. It takes 21 bits to address the two million entries, plus at least one bit to indicate a hit. In matching the final character, we indicate a hit and return to a lower state, the longest suffix of the matching state. In all cases where a match fails, the transition is to the longest suffix of what has been matched.

In the case of DNA, with an alphabet size of four, it is also possible to store the database with four bases per byte, for a 256 character alphabet. This increases the data transfer rate by a factor of four, but it also changes the FSA size. We reduce the length of the query subsequences by a factor of nearly four, but we still increase the number of table entries per database character by a factor of 64. For small FSA on a slow processor, like BLAST uses, this makes sense. If our query substrings were randomly distributed, it would not. However, our query substrings are not statistically independent, and in fact, the substitution/insertion/deletion model gives us many query substrings that differ in only one position. This reduces the number of states required, so it may turn out to be useful. In a hardware implementation, we do better by including more processors. If the processor can run faster than the disk transfer rate, we can unpack the data after it comes off the disk. This would, for example, allow us to use the significantly faster cycle time of static RAM relative to dynamic RAM to our advantage.

6 Finite State Automata as Filters

We consider the Dynamic Programming calculation as the preferred way to score sequence matches. The goal now is to separate likely candidates from unlikely ones. A filter that would reduce the number of comparisons by a factor of ten in the following stage would seem useful, yet practical. If we allow random strings through with a 10% probability, and assume that the real signal is well below this, we should nearly achieve this goal. If we assume DNA with an alphabet size of four, and also assume that the base usage is randomly distributed, we can calculate statistically how many hits we should get with different query substring lengths and numbers of substitutions/insertions/deletions. For the example, each of the 48265 length 16 strings should randomly match one in 4^{16} positions in the database. With a 10^9 database size, or nearly 4^{15}, each of the query strings has about one in four chance of matching. We would like an entire 1000 base sequence to have about a one in ten match rate. One possibility is to increase the length of our query substrings.

Table 1 shows the results if we allow up to one substitution, insertion, or deletion in different length (l) substrings of a 1000 character query. The final column shows the expected number of matches of a random string in a four character alphabet matching against a 10^9 character database.

We can see that 10% is somewhere between 25 and 26 character query substrings, and nearly six million states needed in the FSA. With strings this long, though, one error in 26 may not be enough. If we increase the allowed number of errors, then we need even longer query substrings to maintain the 10% hit rate.

As an additional complication, we should remember that our query strings are not necessarily statistically independent. With the one error allowance, many strings

l	n	e	t	f	x
10	991	91	90181	901810	86003303.53
11	990	100	99000	1089000	23603439.33
12	989	109	107801	1293612	6425440.31
13	988	118	116584	1515592	1737236.98
14	987	127	125349	1754886	466961.41
15	986	136	134096	2011440	124886.63
16	985	145	142825	2285200	33254.04
17	984	154	151536	2576112	8820.56
18	983	163	160229	2884122	2331.64
19	982	172	168904	3209176	614.47
20	981	181	177561	3551220	161.49
21	980	190	186200	3910200	42.34
22	979	199	194821	4286062	11.07
23	978	208	203424	4678752	2.89
24	977	217	212009	5088216	0.75
25	976	226	220576	5514400	0.20
26	975	235	229125	5957250	0.05
27	974	244	237656	6416712	0.01
28	973	253	246169	6892732	0.00
29	972	262	254664	7385256	0.00

The required FSA size is approximately proportional to the total number of characters in the query substrings. We tabulate for different lengths (l) the number of substrings in a 1000 character query (n), the number of cases of each with no errors, or a single substitution, insertion, or deletion using an alphabet size of four (e). Also, the total number of query substrings (t), and total number of characters in those substrings (f). The final column (x) is the expected number of times one of the query subsequences should match in a 10^9 character database, assuming they are statisically independent.

Table 1

are very similar. In any case, to make a useful filter, we must do better than match a single substring with up to one error.

7 A Two Level Finite State Automaton

In order to implement approximate string matching in a Finite State Automaton, we need a vary large number of states. However, many of the states end in a similar result. One possible way to get around this, and allow for a more reasonable memory size, is to drive a second FSA from the output of the first. Suppose we want to match all substrings of length 24 with up to three mismatches. We could generate a FSA to match all substrings of length six with up to three mismatches, and to output a value indicating how many mismatches it found. Then a second FSA matches the patterns in the output of the first. If we have an exact match, the first FSA will generate a series of exact match states. We now have to match four exact match states each six positions apart. There are still a large number of states to match, but the large number of strings with a given number of mismatches will all map into the same state in the output of the first stage.

To see how the number come out, we calculate the case just described. For a DNA alphabet size of four, strings of length six, and up to three substitutions, insertions, or deletions, including the first and last characters when appropriate, the results are shown in table 2. A total of 21065 strings of length six are needed for the first FSA.

For the second FSA we have an alphabet size of 21, the states tallied, plus the "none of the above" state, and strings of length up to 22. We need up to 19 for the exact match, and up to 22 to find the three insertions case. We need to distinguish substitutions, insertions, and deletions to match up the different sub-matches. The result, though, will be that the second FSA is even larger than a single FSA would be.

If we really need to detect all such matches, and only such matches, that is what would be required. But for our filter application, we can use a more statistical method. An FSA that finds six or eight character substrings will find many more of them in a reasonably similar sequence. In a region of a long exact match, it will continuously find matching substrings. In a long approximate match, we will have many consecutive single error matches. We need, then, a way to statistically recognize a good match from the output of a FSA matching smaller substrings. We implement the first FSA to output only the number of errors, zero, one, two, three, or "more", where "more" is too many to be useful.

We implement the second FSA similar to an accumulator, where its state would rise in high match regions, and fall in low match regions, accumulating match scores. If it reached a sufficiently high state, either due to an exact match, or a longer, but less exact match, it would signal the hit. This is a little similar to the way dynamic programming algorithms accumulate scores, though more statistical. It is different than dynamic programming in a special way: at each position it does not destinguish which query substring matched, only that one did. While this would be a disadvantage to dynamic programming, it may be an advantage to us. There are many sequences of marginal similarity that we would otherwise miss. If the query substrings are long enough, they should represent biological features, even if we wouldn't otherwise recognize them. Doing this well implies understanding the details of the sequence

Match condition	Cases	Strings
Exact match	1	1
One substitution	6	18
One deletion	6	6
One insertion	5	20
Two substitutions	30	90
Two deletions	30	30
Two insertions	20	320
Three substitutions	120	3240
Three deletions	120	120
Substitution and Insertion	30	360
Substitution and Deletion	30	90
Substitution and Insertion	30	360
Substitution and two deletions	120	360
Two substititions and deletion	120	1080
Substitution and two insertions	120	5760
Two substitution and insertion	120	4320
Insertion and deletion	30	120
Insertion and two deletions	100	400
Two insertions and deletion	80	1280
Insertion substutution deletion	125	3000

For a six character string, we tabulate the number of cases of substitutions, insertions, deletions, and combinations. The numbers get large very fast, and FSA for these cases must be considered carefully. These numbers are approximate, as they don't include degeneracy in the original sequence, but give an idea about how the query space increases with increasing allowable errors.

Table 2

better than I have described here. The important point, though, is that for longer queries we can use a more statistical approach to the scoring, and still filter what we need to filter.

The biology of approximate matches is a little difficult to describe, but maybe another example will make it more obvious. Imagine an FSA to recognize english words. For an exact answer, one should include an entire dictionary, but realizing the non-uniform distribution of letters and letter groups, one could score based on these groups. Using digraph (two letter) or trigraph (three letter) frequencies in a FSA would recognize possible english words with a high probability. Protein sequences can be similarly recognized by groups of amino acids, even if the groups are in a different order. It is this feature that FSA can find though dynamic programming cannot.

8 Counting FSA States

Until this point, only the total query size was used as a measure of FSA size. Here, the calculation gets more detailed. With a large number of query substrings, the lower FSA states will be well populated. With an alphabet size of four, and the FSA expanding like a tree, the first level will have four states, the second level 16, and the third will have 64. In the terminal branches, the number of states will equal the number of query substrings, each one indicating a hit. In between, it transitions from the saturated lower states, to the sparse terminal states. It is these transition states where the FSA spends most of the time during a search.

For different levels of the tree, a four character alphabet and 10000 query substrings, the numbers are shown in table 3.

The table shows, for each level i of the tree, l the number of states at that level, r the probability of that string matching a random string of that length, and p the probability of matching a string of that length and *not* matching a longer string. This last column gives the probability that the FSA will be at this level of the tree, at an average point during the search.

For an alphabet size of a, the number of possible states at level i is a^i, and there won't be more than the number of query subsequences. If statistically independent, the fraction not used is $(1-a^{-1})^j$ Where j is number of query subsequences per state at the previous level. That is, at each branch of the tree at level i we have j query subsequences to divide up among a new branches. Then l is this fraction multiplied be a multiplied by the number of states at the previous level. Next, the probability p of being at or past level i is la^{-i}, the number of states divided by the number of possible states. The incremental probability, Δp is $p_i - p_{i+1}$, the probability of being at least at level i minus the probablility of being higher than level i.

9 Scaling Laws for FSA Processors

With the ability to put multiple processors on one board, the balance between the size of each processor and the number of processors becomes important. For a fixed board size or fixed cost, we would like to know how many of what sized processors to use. High overall processing speed is achieved by having many processors all running

i	l	p	Δp
1	4	1	0
2	16	1	0
3	64	1	0
4	256	1	1.32e-05
5	1024	0.999987	0.060238
6	3849	0.939748	0.445070
7	8105	0.494678	0.346872
8	9687	0.147806	0.109828
9	9956	0.037978	0.028448
10	9994	0.009531	0.007147
11	9999	0.002384	0.001788
12	10000	0.000596	0.000447
13	10000	0.000149	0.000112
14	10000	3.73e-05	2.79e-05
15	10000	9.31e-06	6.98e-06
16	10000	2.33e-06	1.75e-06
17	10000	5.82e-07	4.37e-07
18	10000	1.46e-07	1.09e-07
19	10000	3.64e-08	2.73e-08
20	10000	9.09e-09	6.82e-09
21	10000	2.27e-09	1.71e-09
22	10000	5.68e-10	4.26e-10
23	10000	1.42e-10	1.07e-10
24	10000	3.55e-11	2.66e-11
25	10000	8.88e-12	6.66e-12
26	10000	2.22e-12	1.67e-12

For each level (i) of the FSA tree for alphabet size four, and 10000 query substrings, the number (l) of states there are likely to be, and the probability of being at least at this level. The final column, Δp, indicates the probability of being at this level and not a higher level.

Table 3

at the same time. Making one large processor, is inconvenient. Some parts, including addressing logic, get bigger as the logarithm of processor size. For result collection reasons, it is more convenient to have the processor size close to the size for a single query sequence, of approximately 1000 bases.

If we make the processor too small, we are limited by the number we can get on a board, and the overhead in board area and cost per processor. The control logic should be done using programmable logic[X96]. Though a simple PAL may be enough, though something a little more complex would probably be useful. In any case, the control logic should be a single chip, and the board area that this takes up will be the most important parameter determining the overall logic density.

Toshiba makes a 64k by 32 bit synchronous static RAM with a 15ns cycle time [TOS98]. This could be clocked at 66 MHz, and hold a 65,536 state FSA. The package itself is 22mm by 16mm. If the control logic was a similar size, and the necessary space between chips added, we need about 10cm^2 of board area.

Samsung makes 128k by 36 bit and 256k by 18 bit synchronous static RAMs with a 6ns cycle time.[SAM98] The access time, the time between the address being clocked in and the data being clocked out is about one half the cycle time, so some time is available for the control logic. Though it will be very difficult to keep up with a 166 MHz clock.

Samsung also makes a 16M by 16 bit synchronous dynamic RAM, with a 72ns read cycle time. (A read cycle requires 9 cycles of a 125 MHz clock.) While the cycle time is much slower than the static RAM, it is closer to the rate we are likely to get data into the processor array. The much larger FSA size will allow overall a greater throughput. As this RAM is only 16 bits wide, two will be required to be able to address all the states. The package size of 10mm by 22mm, a little more than half the SRAM sizes, allows again for about a 10cm^2 processor unit. The processor runs 12 times slower than the SRAM processor, but allows 64 or 128 times the number of states. Except for the extra difficulty of designing for DRAM, this is certainly worth using.

The physical size of a RAM package depends very little on the number of bits stored. Packaging technology is keeping up with current RAM densities, and the silicon size is growing very slowly, approximately logarithmically with the number of bits. With a given size for the control logic and RAM package, what is the optimal number of RAM units to use per control unit?

If the control chip area and RAM chip area are about equal, and we use this size as our unit area, we want to maximize the number of query sequences multiplied by the number of states (input characters) per second we can process. The required number of states is about equal to the number of query sequences per processor. The area for the control logic increases with the required number of address bits, again logarithmically with RAM size.

With the size and speed of the RAM factored out, the number of states per unit area is proportional to $n/(1+n)$, increases asymptotically with increasing n. The number of address lines, and things proportional to the number of address lines increase as $\log n$, so that $n/((1+n)\log n)$ is a better measure, which still increases with increasing n. However, generating the FSA also gets more difficult with increasing n. It is because of this, and simplifying result collection, that we size the RAM to the largest RAM we can get with the smallest number of chips with sufficient data output lines.

10 Result Collection

One of the easiest details to overlook is the collection and storage of result data. If this becomes a bottleneck, it will limit the speed of the entire system. In this case, we need to at minimum know which query sequence matched which database sequence. Optionally, we would know where in the query or database the match was found. As a filter, this additional information is less important.

To find query and database sequence information, we must have an indication of sequence boundaries stored in the database. If we allow only one query per FSA processor, we only need to know which processor detected the hit. We should then latch this state until the next database sequence, avoiding multiple records of the same match.

While the goal is that 10% of the sequences will have hits, in some cases it could approach 100%, and hit collection should be designed to tolerate this case. We could, then, require millions of hits per query per search. Most economical, is to have a small number of hit processors for the entire array. We then have to either buffer hits, or store the entire hit record on each cycle. Buffering requires FIFO (First In First Out) memory in the result path. Storing the entire record, with only one bit per processor, means storing a bit vector at each hit signal. If we allow only one hit per sequence per processor, then we can store the bit vector, one bit per processor, at the end of each database sequence for which we have at least one hit, along with the database sequence number. This is a reasonable compromise in memory required for storing hits.

11 An Implementation Detail

While many implementation details should be left up to the system designer, there is one very interesting one that I describe here. In any technology, dynamic RAMs are much larger in bits stored, than static RAMs, for a similar silicon area and price. In the RAM array, it takes six transistors for a traditional static RAM cell, but only one for a dynamic RAM cell, so it is interesting to consider a design with dynamic RAM. To keep a dynamic RAM refreshed, it is necessary that every value of some of the low order bits be accessed every few milliseconds. RAM is normally implemented as a square array of data cells. In a DRAM read cycle, a row of bits are destructively read out, then stored back into the array. The column address then selects the appropriate bit from the row. It is the read and write back that refreshes the stored charge in the entire row, which happens on any read or write cycle. Normally, processors cannot be depended on to generate addresses sufficiently randomly to rely upon this to refresh the data. Dynamic RAM controllers will add special refresh cycles to guarantee that the data is refreshed in time. But in a processor that naturally cycles through the array, video displays being a common example, this is not necessary.

In designing a Finite State Automaton, it is possible to cycle the low order bits, assuming a little randomness to the input stream. To use real numbers, a certain 256 Megabit DRAM requires each of 8192 rows to be refreshed every 64ms.[SAM98B] If the cycle time is close to 64ns, that means that each row must be accessed every one million cycles. A FSA search processor will normally have some states that it spends much of its time in. Table 3 shows this for a specific combination of parameters. If we take this small number of states, and arrange a succession of equivalent states for them, which cycle the appropriate address bits, it should work. For the case in Table 3, there are 3849 states that together represent 47% of the cycles. If we replace each of these states by ten or more equivalent states, and distribute the addresses among these states, it should be possible to cover the 8192 states needed for the refresh condition.

It may take a fair number of states to do this, but with an 8 million state machine, there should be states to spare.

If this can't be depended upon, it would also be possible to add extra data to the database to insure randomness at the appropriate point. Of course, one could always implement standard refresh logic in the controller.

12 Generating the Finite State Automaton

The ability to process large FSA fast requires the ability to generate them fast. Once we have a list of query sequence fragments, including ones with substitutions, insertions or deletions, a single FSA is generated from them.

With the four character DNA alphabet, we generate a quaternary (base 4) tree for the FSA. To do this, we read in the query sequences and follow through the states of the current FSA. When we try to take a branch with no successor state, we generate a new state and fill all entries with zero, add the branch from the previous state to the new state, and continue on through the sequence.

After we have generated the tree, there will be many states left still with no successor.[W87] It is necessary to back fill these states, to point to the state that the automaton would be in if that state didn't exist. Consider the FSA to match only the query `ACGTACGC`: The states, in succession, will have matched `A`, `AT`, `ACG`, `ACGT`, `ACGTA`, `ACGTAC`, `ACGTACG`, before reaching the final tt ACGTACGC. Now, consider the input `ACGTACGT`. Before the final `T` the FSA will be in the `ACGTACG` state, attempting to match the final `C`. When the `T` is read, it must return to the `ACGT` state, the state matching the longest suffix of the current input.

To do this, an algorithm devised by Gish[Gish] for use in BLAST[AG90] is used. This algorithm starts from the root state and considers every branch that leaves the root state. Each adds onto a circular queue of *from* and *to* states, the branch from the root state to its successor state. Then the queue of *from* and *to* states is then processed. For each branch out of the *to* state that is still zero, we replace it with the corresponding branch out of the *from* state. This does exactly what is needed: it branches to the same place we would have gone if the current state didn't exist. The FIFO (first in first out) queue is important here. The next state must be the state representing the longest match to the input stream. As the tree is traversed, deeper states, representing longer matches, get filled first. The scan and fill process is very fast and executes in time linear in FSA size.

If we are generating a two million state FSA from an input stream, we should consider how fast this process is. With current size machines, we should be able to fit the whole table in processor real memory. In the tree building phase, for each input character, we need to check the current table entry, add a new state if it doesn't exist, and then move to the next state. For the backfill process, we need to progress through the tree as states are added to and removed from the FIFO queue. The queue depth could approach the number of states in the FSA, and each queue entry needs two state pointers.

The generate and backfill algorithm should be fast enough to keep the system running. During a search, the processor should be able to stream data at 16 million characters per second. The time needed to generate and load the FSA should be less than the search time. With upcoming genome projects expected to generate 30 gigabase data sets within three years, there should be enough data to keep the system running.

If our search time is on the order of hours and we need hundreds of FSA's generated in that time, we must generate them in minutes. We should be able to generate two million states in times the order of seconds on 100 MHz processors. Writing the generated FSA out to disk is the slowest part.

For the dynamic RAM version using statistical refresh, we want multiple copies of the more commonly occupied states. If we generate a complete tree for the first levels, with multiple identical copies of the level common states distributed through the low order address bits, and then add new states onto this, we should have a good start.

13 Conclusion

A large array of Finite State Automaton processors can be built for a reasonable price. This array can be used to rapidly search a database for some set of query sequences, and to store information related to the query sequences found. In some cases, this may be enough, otherwise, it can be used as input to a more detailed search. It should increase the value of the more detailed search by concentrating the useful sequences.

C code fragment to backfill states in a FSA

```
q=0;
for(i=0;i<ALPHABET;i++) {
   j=fsa[i];
   j &=  ACCEPT;
   if(j==0) continue;
   fifo[q].from=0;
   fifo[q].to=j;
   q++;
   }
head=q-1;
tail=0;
while(q>0) {
   from=fifo[tail].from;
   to=fifo[tail].to;
   tail=(tail+1)%(state+1);
   q--;
   for(i=0;i<ALPHABET;i++) {
      j=fsa[from+i];
      k=fsa[to+i];
      if(!k) {
         fsa[to+i]=j;
         continue;
         }
      if(k & ACCEPT) {
         fsa[to+i]=j | ACCEPT;
         continue;
         }
      if(!(j & ACCEPT)) {
         head=(head+1)%(state+1);
         q++;
         fifo[head].from=j;
         fifo[head].to=k;
         }
      }
   }
```

Figure 1: After all the query sequences are added to the FSA it is necessary to backfill it. Each link that is not part of a query sequence must point back to the state that the FSA would be in with the same input if the current state did not exist. The significant feature of this algorithm is the FIFO queue of states to be done.

References

[AG90] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J., *J. Mol. Biol.* **215**, 403-410 (1990).

[CH91] Chow E.T., Hunkapiller T., Peterson J.C., Zimmerman B.A., Waterman M.S., A systolic array processor for Biological Information Signal Processing. Proc. of International Conference on Supercomputing (ICS-91) June 17-21, 1991.

[Gish] Personal communication.

[G82] Gotoh, O., An improved algorithm for matching biological sequences, J. Theor. Biol., **162**, 705-708 (1982).

[HU79] Hopcroft, J.E., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley, Reading, Massachusetts.

[KA90] Karlin, S., Altschul, S.F., *Proc. Natl. Acad. Sci. USA* **87** 2264-68 (1990).

[KA93] Karlin, S., Altschul, S.F., *Proc. Natl. Acad. Sci. USA* **90** 5873-7 (1993).

[LL85] Lipton, R.S., Lopresti, D., A Systolic Array for Rapid String Comparison, 1985 Chapel Hill Conference on VLSI (1985)

[NW70] Needleman, S.B., Wunch, C.D., A general method applicable to the search for similarities in the amino acid sequence of proteins, *J. Mol. Biol.*, **48**:443-453 (1970).

[SAM98] 256K x 18 bit Synchronous Pipelined Burst SRAM, KM718V889, Samsung Electronics, (1997).

[SAM98B] 4M x 16bit x 4 Banks Synchronous DRAM, KM416S16230A, Samsung Electronics, (1997).

[TOS98] 65,536 word by 32 bit Synchronous Pipelined Burst Static RAM, TC55V2325FF-7, Toshiba Corporation (1998).

[W87] Wood, Derick: Theory of Computation. Harper & Row, New York, New York.

[X96] Xilinx, Inc., *The Programmable Logic Data Book,* Xilinx, Inc., 1996

Dynamic Programming for Reduced NFAs for Approximate String and Sequence Matching[1]

Jan Holub

Department of Computer Science and Engineering
Czech Technical University
Karlovo nám. 13, 121 35 Prague 2, Czech Republic
phone: (++420 2) 2435 7287, fax: (++420 2) 298098

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

Abstract. We present a new simulation method for the reduced nondeterministic finite automata (*NFA*s) for the approximate string and sequence matching using the Levenshtein and generalized Levenshtein distances. These reduced *NFA*s are used in case that we are interested only in all occurrences of a pattern in an input text such that the edit distance between the pattern and the found strings is less or equal to a given k and we are not interested in the values of these edit distances. The presented simulation method is based on the dynamic programming.

Key words: approximate string and sequence matching, simulation of nondeterministic finite automata, Levenshtein distance, generalized Levenshtein distance, dynamic programming

1 Introduction

Given a string $T = t_1t_2 \dots t_n$ over an alphabet Σ, a pattern $P = p_1p_2 \dots p_m$ over the alphabet Σ, and an integer k, $k \leq m \leq n$. The approximate string matching is defined as a searching for all occurrences of pattern P in text T such that edit distance $D(P, X)$ between pattern P and string $X = t_it_{i+1} \dots t_j$, $0 < i \leq j \leq n$, found in the text is less than or equal to k. The approximate sequence matching is defined in the same way as the approximate string matching, but any number of symbols can be located between the occurrences of two adjacent symbols of the pattern in the text. In this paper we consider two types of distances called the Levenshtein distance and the generalized Levenshtein distance.

The Levenshtein distance $D_L(P, X)$ between strings P and X not necessarily of the same length is the minimum number of edit operations *replace* (one character is replaced by another), *insert* (one character is inserted), and *delete* (one character is removed) needed to convert string P to string X. The generalized Levenshtein distance $D_G(P, X)$ between strings P and X not necessarily of the same length is the minimum number of edit operations *replace*, *insert*, *delete*, and *transpose* (two adjacent characters are exchanged) needed to convert string P to string X.

[1]This research was partially supported by grant 201/98/1155 of the Grant Agency of Czech Republic and by internal grant 3098098/336 of Czech Technical University.

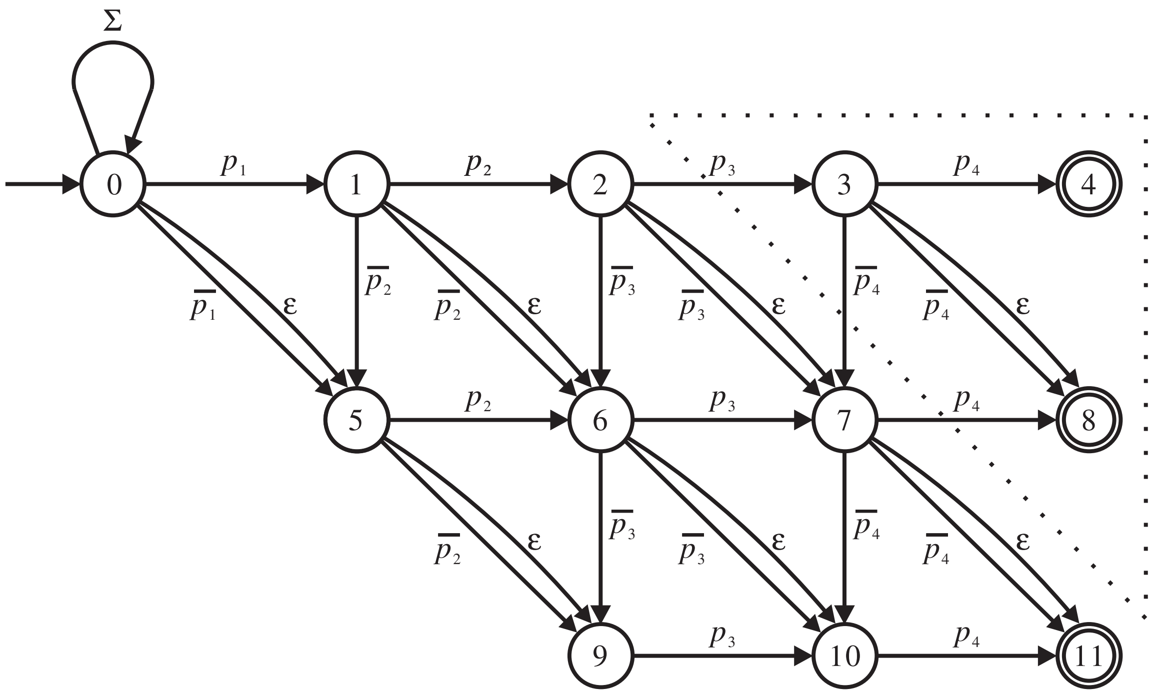


Figure 1: *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 2$).

The nondeterministic finite automaton (*NFA*) for the approximate string matching using the Levenshtein distance has been presented in [Mel96, Hol96]. In the *NFA* there is for each edit distance l, $0 \le l \le k$, one level of states. An example of such *NFA* for $m = 4$ and $k = 2$ is shown in Figure 1[2].

There are known two algorithms for the approximate string matching for which there was shown [Mel96, Hol97] that they simulate the run of the *NFA* for the approximate string matching. The first method is Shift-Or algorithm [BYG92] and its variations — Shift-Add [BYG92] and Shift-And [WM92]. The second method is the dynamic programming [Sel80, Ukk85].

2 Reduced *NFA*s

If we are interested only in all occurrences of the pattern in the text with the edit distance less or equal to k, and we do not want to know the edit distance between the found string and the pattern, we can remove such states from the *NFA* for the approximate string matching that are needed only to determine the edit distance of the found string [Hol96]. Such states are bordered by the dotted line in Figure 1. The resulting *NFA* is shown in Figure 2 and has only one final state that represents that the pattern has been found with the edit distance less or equal to k.

[2]Symbol $\overline{p}_j$, $0 < j \le m$, represents $\Sigma - \{p_j\}$ in figures.

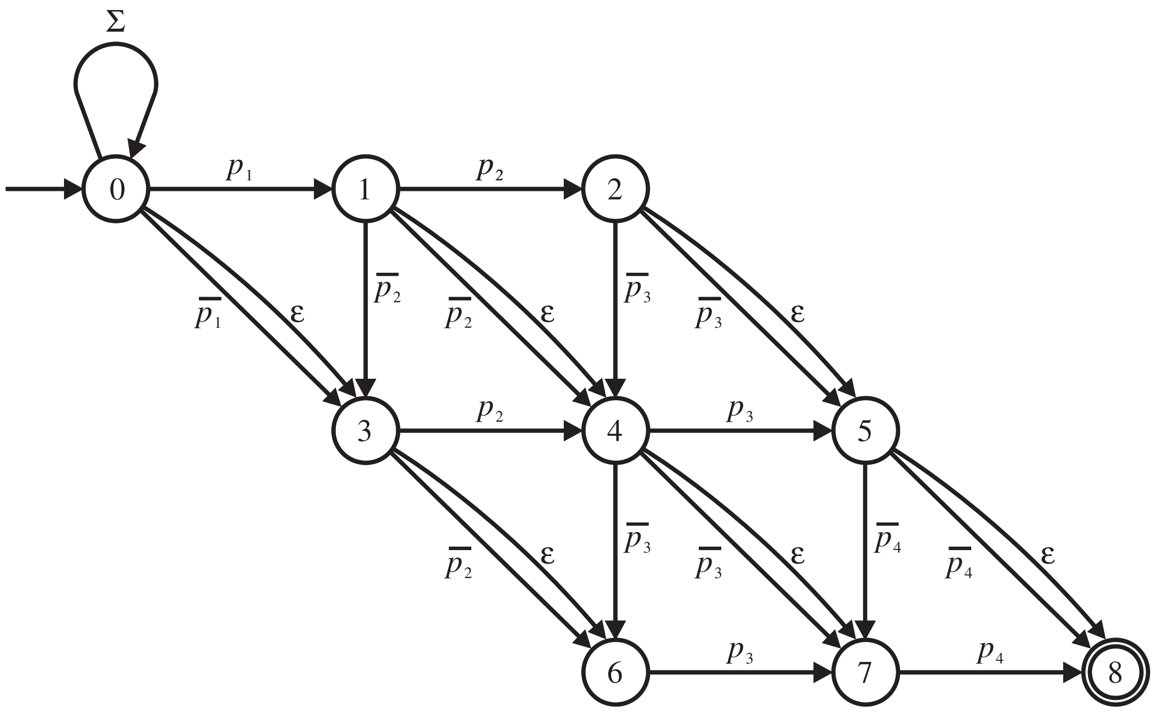


Figure 2: Reduced *NFA* for the approximate string matching using the Levenshtein distance ($m = 4$, $k = 2$).

The modification of Shift-Or algorithm for the reduced *NFA*s was presented in [Hol96] and the modification of the dynamic programming is discussed in the following sections.

3 Dynamic Programming

The dynamic programming [Sel80, Ukk85] computes in each step i of the run of the *NFA* ith column of matrix D which is of size $(m \times n)$; one element of the column is for each depth of the *NFA* and contains the number of level of the highest active state of this depth. If there is no active state in this depth, then the element contains the number of the level not existing in this depth. Since each *NFA* for the approximate string matching has $m + 1$ depths, it needs space $\mathcal{O}(m)$ and runs in time $\mathcal{O}(mn)$.

Since last k depths of the *NFA* do not have states on all $k + 1$ levels of the *NFA*, this method is not suitable for the reduced *NFA*s for the approximate string matching. Instead of having one element of the column for each depth of the *NFA* we have one element for each diagonal of the *NFA*; these diagonals are formed by the ε-transitions and are of the same length. If any state on a diagonal is active, then all states located lower on this diagonal are also active because of ε-transitions. Therefore in the element for each diagonal l, $0 \leq l \leq m - k$, we store only the number of the level of the highest active state on diagonal l. In this way we get for each step i, $0 \leq i \leq n$, of the run of the *NFA* the column $D_i = d_{0,i}, d_{1,i}, \ldots, d_{m-k,i}$ of length $m - k + 1$. Each element of the column can contain a value ranging from 0 to $k + 1$, where value $k + 1$

represents that there is no active state on the corresponding diagonal. The formula for computing columns D_i is as follows:

$$
\begin{array}{lcll}
d_{0,i} & := & 0, & 0 \le i \le n \\
d_{j,0} & := & k+1, & 0 < j \le m-k \\
d_{j,i} & := & \mathbf{min}(k+1, & \\
 & & \quad g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}, & \textit{delete \& match} \\
 & & \quad \mathbf{if}\ p_{d_{j,i-1}+j+1} \neq t_i & \\
 & & \quad \mathbf{then}\ d_{j,i-1}+1 & \textit{replace} \\
 & & \quad \mathbf{else}\ k+1, & \\
 & & \quad \mathbf{if}\ p_{d_{j+1,i-1}+j+2} \neq t_i & \\
 & & \quad \mathbf{then}\ d_{j+1,i-1}+1 & \textit{insert} \\
 & & \quad \mathbf{else}\ k+1), & 0 < j < m-k, 0 < i \le n \\
d_{j,i} & := & \mathbf{min}(k+1, & \\
 & & \quad g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}, & \textit{delete \& match} \\
 & & \quad \mathbf{if}\ p_{d_{j,i-1}+j+1} \neq t_i & \\
 & & \quad \mathbf{then}\ d_{j,i-1}+1 & \textit{replace} \\
 & & \quad \mathbf{else}\ k+1), & j = m-k, 0 < i \le n
\end{array}
\qquad (3)
$$

The first line in the formula says that the initial state lying on the 0th diagonal of the *NFA* is always active because of its self-loop.

The second one says that at the beginning of the searching there is no active state on diagonals l, $0 < l \le m-k$, because there is no initial state on such diagonals.

Part $g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}$ represents *match* and *delete* transitions. The *match* is represented by the horizontal transitions and edit operation *delete* is represented by the diagonal ε-transitions in Figure 2. An implementation of *match* transition is simple — if the state on diagonal $j-1$ and on level $d_{j-1,i-1}$ is active and horizontal transition leading from this state is labeled by symbol t_i, then the state on diagonal j and on level $d_{j-1,i-1}$ becomes active. For an implementation of *delete* transition we have to search for the state on diagonal $j-1$ and on level l, $d_{j-1,i-1} \le l \le m-k$, such that there is a *match* transition labeled by input symbol t_i leading from this state. In order to find such state in the constant time we have to use auxiliary matrix G in which there is for each position r in pattern P and input symbol t_i the number r', $0 \le r'$, such that $p_{r+r'} = t_i$ where r' is the lowest possible. If there is no such position, then $r' = k+1$. Since the value of $d_{j-1,i-1}$ can be $k+1$ and the maximum number of diagonal, into which there lead *match* transitions, is $m-k$, the maximum position for which a value of matrix G is required is $m-k+k+1 = m+1$. Therefore the matrix has to be of size $(m+2) \times |\Sigma'|$ where $\Sigma' \subseteq \Sigma$ is the alphabet used in pattern P. The formula for computation of matrix G is as follows:

$$
\begin{array}{lcll}
g_{j,a} & := & \mathbf{min}(\{k+1\} \cup \{(l \mid p_{j+l} = a, 0 \le l) \text{ or} & \\
 & & \quad (k+1 \mid \text{if there is no such } l)\}), & 0 < j \le m, a \in \Sigma \\
g_{m+1,a} & := & k+1, & a \in \Sigma
\end{array}
\qquad (4)
$$

Number $d_{j-1,i-1} + j$ gives the position of symbol $p_{d_{j-1,i-1}+j}$ in the pattern which is used as a label of the *match* transition leading from the highest active state on diagonal $j-1$ to a state on diagonal j. Therefore $g_{d_{j-1,i-1}+j,t_i} + d_{j-1,i-1}$ gives the level of the highest active state on diagonal j that has arisen by using *match* transition to each active state on diagonal $j-1$.

Part $d_{j,i-1}+1$ represents *replace* transition. In Figure 2, edit operation *replace* is represented by the diagonal transition labeled by symbol $\overline{p}_{d_{j,i-1}+j+1}$ mismatching symbol $p_{d_{j,i-1}+j+1}$. To implement *replace* transition it is only needed to move the highest active state on diagonal j to the next lower position on the same diagonal. Since $d_{j,i-1}$ can reach $k+1$ the value of expression $d_{j,i-1}+j+1$ can be greater than m and in that case $p_{d_{j,i-1}+j+1}$ would give undefined value. To solve this problem we can add some **if** statements but it increases the time of the computation. The better solution is to put some symbols, that are not in input alphabet Σ, at positions $m+1$ and $m+2$ of the pattern — for example symbol $\langle$*end of string*$\rangle$.

Part $d_{j+1,i-1}+1$ represents *insert* transition. In Figure 2, edit operation *insert* is represented by the vertical transition also labeled by mismatching symbol $\overline{p}_{d_{j+1,i-1}+j+2}$. The active state on diagonal $j+1$ and on level $d_{j+1,i-1}$ moves to level $d_{j+1,i-1}+1$ on diagonal j.

From these transitions we get minimum in order to obtain the highest active state on each diagonal. An example of matrix G for pattern $P = adbbca$ is shown in Table 1 and the process of searching for pattern $P = adbbca$ with at most $k = 3$ errors in text $T = adcabcaabadbbca$ is shown in Table 2.

G	a	b	c	d	$\Sigma - \{a,b,c,d\}$
1	0	2	4	1	4
2	4	1	3	0	4
3	3	0	2	4	4
4	2	0	1	4	4
5	1	4	0	4	4
6	0	4	4	4	4
7	4	4	4	4	4

Table 1: Matrix G for pattern $P = adbbca$ and $k = 3$.

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	4	0	1	1	0	1	2	0	0	1	0	1	2	2	3	0
2	4	4	0	1	2	1	2	3	4	1	2	0	1	2	3	4
3	4	4	4	2	3	4	2	3	3	4	3	4	0	1	2	3

Table 2: Matrix D for pattern $P = adbbca$, text $T = adcabcaabadbbca$, and $k = 3$.

Below we also present an algorithm that uses the dynamic programming for the reduced *NFA* for the approximate string matching using the Levenshtein distance. While in Formula 3 there were evaluated the transitions incoming to the diagonals, in this algorithm there are evaluated the outgoing transitions. It simplifies the computation because then there is only one test whether input symbol t_i is a matching symbol. This test is necessary for deciding whether to use only *match* transition or to use *replace*, *insert*, and *delete* transitions.

Algorithm 1
DP for the reduced *NFA* for the approximate string matching using the Levenshtein distance
Input: Pattern $P = p_1p_2 \dots p_m$, text $T = t_1t_2 \dots t_n$, maximum number of differences allowed k.
Output: Matrix D of size $(m-k+1) \times (n+1)$.
Method:

```
d_{0,0} := 0
d_{j,0} := k+1, 0 < j ≤ m-k
for i := 1,2,...,n do
  d_{0,i} := 0                                            /* j = 0 */
  d_{1,i} := g_{1,t_i}       /* delete & match from the initial state *** */
  if p_{d_{1,i-1}+2} = t_i then                           /* j = 1 */
    d_{2,i} := d_{1,i-1}                                  /* match */
  else
    d_{2,i} := min(g_{d_{1,i-1}+2,t_i} + d_{1,i-1}, k+1)  /* delete & match */
    d_{1,i} := min(d_{1,i-1} + 1, d_{1,i})                /* replace */
  endif                                                   /* *** */
  for j := 2,3,...,m-k-1 do
    if p_{d_{j,i-1}+j+1} = t_i then
      d_{j+1,i} := d_{j,i-1}                              /* match */
    else
      d_{j+1,i} := min(g_{d_{j,i-1}+j+1,t_i} + d_{j,i-1}, k+1)  /* delete & match */
      d_{j,i} := min(d_{j,i-1} + 1, d_{j,i})              /* replace */
      d_{j-1,i} := min(d_{j,i-1} + 1, d_{j-1,i})          /* insert */
    endif                                                 /* *** */
  endfor
  j := m-k                                                /* the last diagonal */
  if p_{d_{j,i-1}+j+1} ≠ t_i then
    d_{j,i} := min(d_{j,i-1} + 1, d_{j,i})                /* replace */
    d_{j-1,i} := min(d_{j,i-1} + 1, d_{j-1,i})            /* insert */
  endif
  if d_{m-k,i} < k+1 then
    write("pattern found at position i")
  endif
endfor
```

The first command in the first **for** cycle in the algorithm ($d_{0,0} := 0$) represents the self-loop of the initial state — the highest active state in 0th diagonal is always in level 0 and this is the initial state.

The second command ($d_{1,i} := g_{1,t_i}$) represents the only transition that leads from 0th diagonal which is *match* transition. g_{1,t_i} gives the position l of the pattern, on which t_i is located, or $k+1$ if t_i is not in the pattern. If $l < k+1$, then this position l

is equal to the level of 1st diagonal in which there is the active state that arose by using *match* transition for t_i going from 0th diagonal.

The first **if** statement represents transitions leading from the highest active state on the 1st diagonal. In this case we do not evaluate *insert* transitions because they lead always to 0th diagonal where the initial state is always active. If input symbol t_i is the same as the symbol $p_{d_{1,i-1}+2}$ used as a label of *match* transition leading from the highest active state in 1st diagonal, then we evaluate only this *match* transition ($d_{2,i} := d_{1,i-1}$). If the symbols are different, then we evaluate *delete* and *replace* transitions. For *delete* transition we search for the next occurrence of input symbol t_i in the pattern behind position $d_{j,i-1} + j + 1$ (the number of the diagonal plus the number of the level gives the position in the pattern corresponding to the state on that level of that diagonal). At first we perform *delete* transition (we move the highest active state down in the diagonal) and then we perform *match* transition for input symbol t_i. For *replace* transition we move the highest active state in the diagonal to the next lower position in the diagonal.

In the next **for** cycle the transitions leading from the highest active state of the next diagonals except the last one are evaluated. It is done in the same way as described in the previous paragraph but in addition *insert* transition is evaluated. For this *insert* transition we put the level of diagonal j increased by one to the previous diagonal $j - 1$.

In the last diagonal we evaluate only *replace* and *insert* transitions because *match* transition has no diagonal into which it could lead.

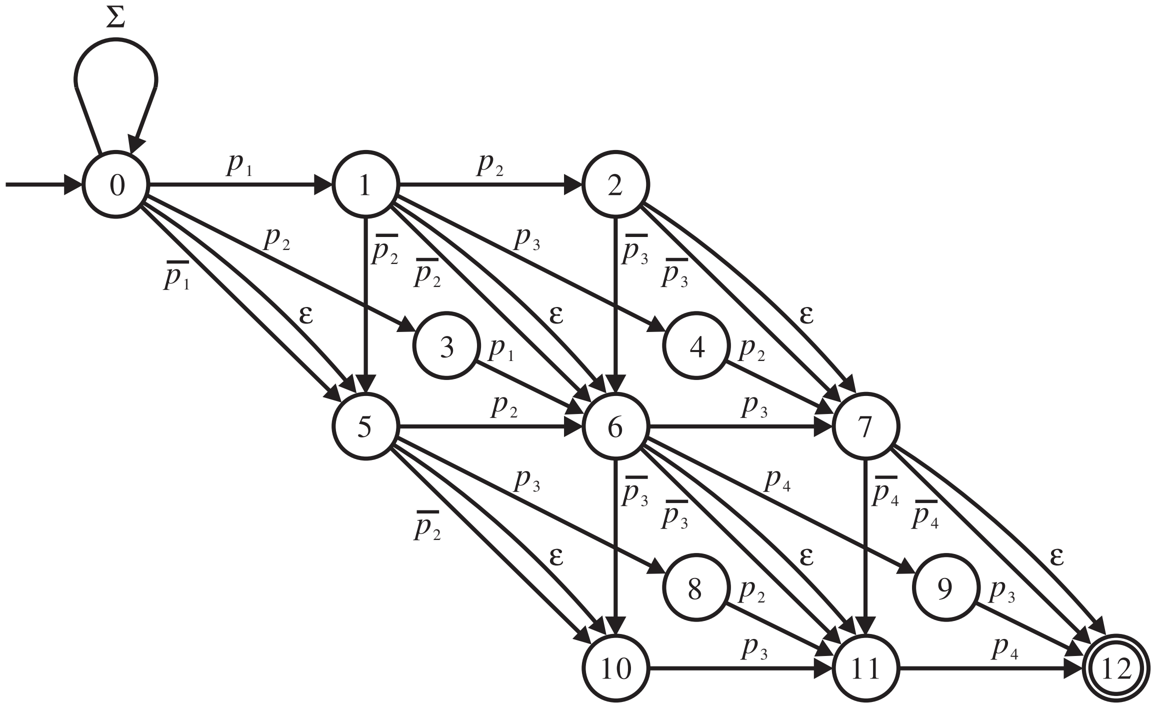


Figure 3: Reduced *NFA* for the approximate string matching using the generalized Levenshtein distance ($m = 4$, $k = 2$).

This method can be also used for the simulation of the run of the reduced *NFA* for the approximate string matching using the generalized Levenshtein distance. An example of such reduced *NFA* for $m = 4$ and $k = 2$ is shown in Figure 3. We have only to add the part representing edit operation *transpose*. In Formula (3), the added part is as follows:

$$\begin{array}{ll} \textbf{if } p_{d_{j-1,i-2}+j+1} = t_{i-1} \textbf{ and } p_{d_{j-1,i-2}+j} = t_i & \\ \textbf{then } d_{j-1,i-2}+1 & \mathit{transpose} \\ \textbf{else } k+1, & 0 < j \leq m-k, 1 < i \leq n \end{array} \qquad (5)$$

And in Algorithm 1, the added part is as follows:

$\textbf{if } \; p_{d_{j,i-2}+j+2} = t_{i-1} \textbf{ and } p_{d_{j,i-2}+j+1} = t_i \textbf{ then}$
$\quad d_{j+1,i} := \textbf{min}(d_{j,i-2}+1, d_{j+1,i})$ /∗ *transpose* ∗/
endif

This part should be inserted into each part of Algorithm 1 where $0 \leq j < m-k$ and $1 < i \leq n$. Such places are behind the lines marked by '∗∗∗'.

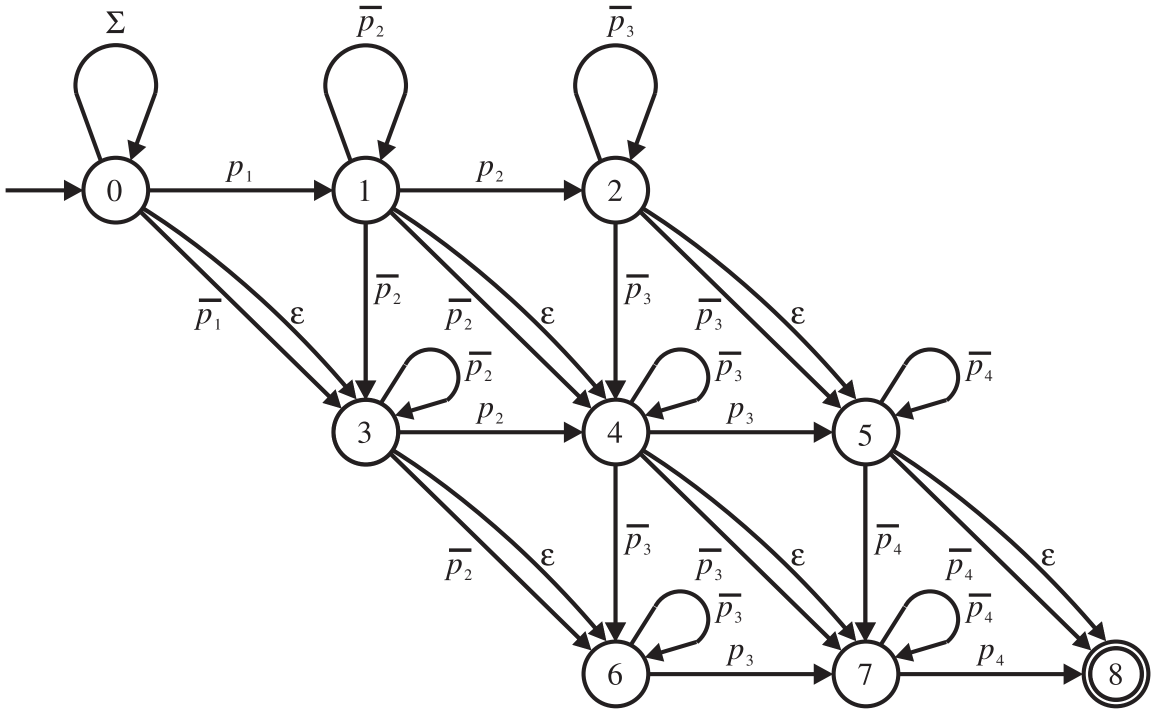


Figure 4: Reduced *NFA* for the approximate sequence matching using the Levenshtein distance ($m = 4$, $k = 2$).

This type of simulation of the reduced *NFA*s can be also used for the reduced *NFA*s for the approximate sequence matching using the Levenshtein and generalized Levenshtein distances [Hol97]. An example of the reduced *NFA* for the approximate sequence matching using the Levenshtein distance for $m = 4$ and $k = 2$ is shown in Figure 4.

To modify the presented algorithm so that it could simulate this reduced *NFA* we have to implement the self-loops in each nonfinal and noninitial state. It can be performed by inserting the following part into Formulae (3) and (3+5) for the approximate string matching.

$$\begin{array}{ll}
\textbf{if } p_{d_{j,i-1}+j+1} \neq t_i & \\
\textbf{then } d_{j,i-1} & \mathit{self-loop} \\
\textbf{else } k+1, & 0 < j < m-k, 0 < i \leq n \\
\textbf{if } p_{d_{j,i-1}+j+1} \neq t_i \textbf{ and } d_{j,i-1} < k & \\
\textbf{then } d_{j,i-1} & \mathit{self-loop} \\
\textbf{else } k+1, & j = m-k, 0 < i \leq n
\end{array} \qquad (6)$$

The presented formulae and algorithm compute whole matrix D but in the practice only two (three for the generalized Levenshtein distance) columns from this matrix are used in each step of the computation.

4 Conclusion

The resulting simulation runs in time $\mathcal{O}((m-k)n + m\mu)$ and needs space $\mathcal{O}(m\mu)$, where μ is the number of different symbols used in the pattern. We can decrease the space complexity by using another implementation of auxiliary matrix G but it increases the time complexity. Our algorithm also uses only one input symbol in each step of computation in case of the Levenshtein distance and two input symbols in case of the generalized Levenshtein distance.

The resulting algorithm has the time bound better than [Sel80, Ukk85] which runs in time $\mathcal{O}(mn)$ and for $k > \frac{m}{2}$ it has also the time bound better (not considering the preprocessing time) than [GP89] which runs in time $\mathcal{O}(kn + m \log \tilde{m})$ where $\tilde{m} = \min(m, |\Sigma|)$.

References

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.

[GP89] Z. Galil and K. Park. An improved algorithm for approximate string matching. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, number 372 in Lecture Notes in Computer Science, pages 394–404, Stresa, Italy, 1989. Springer-Verlag, Berlin.

[Hol96] J. Holub. Reduced nondeterministic finite automata for approximate string matching. In J. Holub, editor, *Proceedings of the Prague Stringologic Club Workshop '96*, pages 19–27, Prague, Czech Republic, 1996. Collaborative Report DC-96-10.

[Hol97] J. Holub. Simulation of NFA in approximate string and sequence matching. In J. Holub, editor, *Proceedings of the Prague Stringology Club Workshop '97*, pages 39–46, Czech Technical University, Prague, Czech Republic, 1997. Collaborative Report DC–97–03.

[Mel96] B. Melichar. String matching with k differences by finite automata. In *Proceedings of the 13th International Conference on Pattern Recognition*, volume II., pages 256–260, Vienna, Austria, 1996. IEEE Computer Society Press.

[Sel80] P. H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1(4):359–373, 1980.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1–3):132–137, 1985.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

Validating and Decomposing Partially Occluded Two-Dimensional Images (Extended Abstract)[1]

Costas S. Iliopoulos[1,2] and James F. Reid[1,3]

[1] Algorithm Design Group, Department of Computer Science
King's College London, London WC2R 2LS, UK.
[2] School of Computing, Curtin University of Technology
Perth, WA 6102, Australia.
[3] Dipartimento di Elettronica e Informatica
Università degli Studi di Padova
Via Gradenigo 6/a, 35131 Padova, Italy.

e-mail: {csi,jfr}@dcs.kcl.ac.uk

Abstract. A partially occluded scene in an image consists of a number of objects that are partially obstructed by others. Validating a partially occluded image consists of generating a sequence of concatenated and possibly overlapping objects that corresponds to the input image. The algorithm presented here validates a two-dimensional image X of size $r \times s$ over a set of k objects of identical size $m \times m$ in $O(mrs)$ time.

Key words: String algorithms, image processing, occlusion analysis, pattern recognition.

1 Introduction

The study of repetitive structures in sequences (strings) plays a key role in information processing and more generally in computer science. This has lead to a generalization of notions concerning repetitions in sequences. The periodicity of a string was the key element in the design of the famous pattern matching algorithm by Knuth, Morris and Pratt, [KMP-77]. A related notion is the one of a cover of a string. A substring w of a string x is called a *cover* of x if x can be constructed by concatenations and superpositions of w. As a result, many sequential and parallel algorithms have been developed concerning the covering of a string. Among the sequential algorithms, Apostolico, Farach and Iliopoulos [AFI-91] solved the problem of computing the shortest cover of a given string, similarly Moore and Smyth [MS-95] solved the problem of computing all the covers of a given string both in linear time. These

[1]C.S. Iliopoulos was partially supported by EPSRC grants GR/F 00898, GR/L 19362 and GR/J 17844, NATO grant CRG 900293, and MRC grant G9115730. J.F. Reid was supported by a Marie Curie fellowship of the European Commission Training and Mobility of Researchers (TMR) Programme.

efficient solutions for string covering problems have applications to DNA sequencing by hybridization, see [DS-96] and [PL-94].

This paper focuses on an application of the string covering techniques to image processing and the analysis of images composed of know objects obstructing each other. Decomposing partially occluded images is a classical problem in computer vision and its computational complexity is exponential. There are many artificial intelligence and neural network solutions to this problem, see for example [BC-94]. Here we present a theoretical study on the analysis of images composed from a given set of objects, where some of the appearing objects may be partially occluded by other ones. Thus we restrict our attention on the occlusion problem by focusing only on discrete images and convex objects, and their efficient solutions are based on the study of the repetitive structures of the input. The results and solutions presented here provide the foundations for practical solutions to this problem. This problem was first approached by only considering one-dimensional images (strings). A linear sequential on-line algorithm was produced by Iliopoulos and Simpson [IS-97] and an optimal parallel version was also produced, see [IR-97].

In the following, we will consider the family of two dimensional images (considered as two-dimensional arrays of strings), that we call *valid images*; given a set of objects $\{S_1, \ldots, S_k\}$ and a special "background" symbol denoted $\#$, an image X of size $r \times s$ is a valid image, if X is iteratively obtained from an initial string Z of size $r \times s$ consisting only of $\#$'s by substituting substrings of Z by some objects S_i, for some i. We will be focusing in designing algorithms for testing two-dimensional images for validity, under restricted sets of objects, i.e. square objects of the same size.

Here we present an algorithm for validating a two-dimensional image X of size $r \times s$ over a fixed number k of objects S_i of equal size $m \times m$ in $O(mrs)$ time.

The paper is organised as follows. In the next section we present the basic definitions for strings and partially occluded images. In Section 3 we describe the data structures and the main techniques used in the algorithm and finally in Section 4 we present our conclusions and open problems.

2 Preliminaries

2.1 String definitions in one and two dimensions

A *string* (or word) is a sequence of zero or more symbols drawn from an alphabet Σ, which consists of a finite number of symbols. The set of all strings over Σ is denoted by Σ^*. The string of length zero is the *empty string* ϵ and a string x of length $n > 0$ is represented by $x_0 x_1 \cdots x_{n-1}$, where $x_i \in \Sigma$ for $0 \leq i \leq n-1$. A string w is said to be *substring* of x if and only if $x = uwv$ for some $u, v \in \Sigma^*$. A string w is a *prefix* of x if and only if $x = wu$ for some $u \in \Sigma^*$; if u is not empty then w is a called a *proper prefix* of x. Similarly, w is a *suffix* of x if and only if $x = uw$ for some $u \in \Sigma^*$; if u is not empty then w is called a *proper suffix* of x. Additionally $prefix_k(x)$ denotes the first k symbols of x and $suffix_k(x)$ denotes the last k symbols of x. The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k. For two strings $x = x_0 \cdots x_{n-1}$ and $y = y_0 \cdots y_{m-1}$ such that $x_{n-i} \cdots x_{n-1} = y_0 \cdots y_i$ for some $i \geq 1$ (that is, such that x has a suffix equal to a prefix of y), the string $x_0 \cdots x_{n-1} y_i \cdots y_{m-1}$ is said to be a *superposition* of x and y.

Alternatively, we may say that x *overlaps* with y. A substring w of x is called a *cover* of x if x can be constructed by concatenations and superpositions of w.

A *two-dimensional string* is an $r \times s$ array of symbols drawn from Σ. We will refer to a two-dimensional string as a *two-dimensional array* or a *two-dimensional image* in the sequel. We represent an $r \times s$ array X by $X[0..r-1, 0..s-1]$. A two-dimensional $p \times q$ array Y is said to be a *sub-array* or a *sub-image* of X if the upper left corner of Y can be aligned with $X[i,j]$, i.e. $Y[0..p-1, 0..q-1] = X[i..i+p-1, j..j+q-1]$, for some $0 \leq i \leq r-p$ and $0 \leq j \leq s-q$. A square $m \times m$ sub-array Y is said to be a *prefix* of X, if Y occurs at position $X[0..m-1, 0..m-1]$. Similarly, Y is said to a *suffix* of X if, Y occurs at position $X[r-m..r-1, s-m..s-1]$.

2.2 Definitions and properties of partially occluded images

In the following we assume that Σ is a finite alphabet of symbols. We denote the symbol $\# \notin \Sigma$ to be a special symbol called the *background* symbol.

Definition 2.1 Let X be a $r \times s$ array called the *image* over the alphabet Σ and let $\mathcal{O} = \{S_1, \ldots, S_k\}$ be a set of $m \times m$ arrays called the *objects* also over Σ. Then the image X is said to be a *valid image* over $\mathcal{O}$ if and only if $X = Z_i$ for some $1 \leq i \leq rs-1$, where

$$Z_0 = \begin{pmatrix} \# & \cdots & \# \\ \vdots & \ddots & \vdots \\ \# & \cdots & \# \end{pmatrix},$$

$$Z_{i+1} = \left(\begin{array}{c|c|c} \mathrm{pref}(Z_i) & \mathrm{sub}(Z_i) & \mathrm{sub}(Z_i) \\ \hline \mathrm{sub}(Z_i) & S_t & \mathrm{sub}(Z_i) \\ \hline \mathrm{sub}(Z_i) & \mathrm{sub}(Z_i) & \mathrm{suff}(Z_i) \end{array}\right)$$

where $S_t \in \mathcal{O}$ for some $t \in \{1, \ldots, k\}$.

The recurrence equalities defined above are said to be the *substitution rules* and the sequence $Z_0, Z_1, \ldots, Z_i$ is said to be the *generating sequence* of the image X over the set of objects $\mathcal{O} = \{S_1, \ldots, S_k\}$. We now construct such a generating sequence for a partially occluded image in the following example.

Example 2.1 Let $\mathcal{O} = \left\{ S_1 = \begin{pmatrix} c & c \\ d & b \end{pmatrix}, S_2 = \begin{pmatrix} a & b \\ b & c \end{pmatrix}, S_3 = \begin{pmatrix} a & d \\ b & c \end{pmatrix}, S_4 = \begin{pmatrix} b & b \\ c & c \end{pmatrix} \right\}$ be the set of objects and let the image be

$$X = \begin{pmatrix} a & b & a & d \\ b & c & b & c \\ c & a & b & b \\ d & b & c & c \\ a & b & d & b \\ b & c & b & c \end{pmatrix}$$

Then X is a valid image over $\mathcal{O}$ with the following generating sequence:

$$Z_0 = \begin{pmatrix} \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \end{pmatrix}, Z_1 = \begin{pmatrix} \underline{a} & \underline{b} & \# & \# \\ \underline{b} & \underline{c} & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \end{pmatrix}, Z_2 = \begin{pmatrix} a & b & \# & \# \\ b & c & \# & \# \\ \underline{c} & \underline{c} & \# & \# \\ \underline{d} & \underline{b} & \# & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \end{pmatrix},$$

$$Z_3 = \begin{pmatrix} a & b & \# & \# \\ b & c & \# & \# \\ c & \underline{a} & \underline{b} & \# \\ d & \underline{b} & \underline{c} & \# \\ \# & \# & \# & \# \\ \# & \# & \# & \# \end{pmatrix}, Z_4 = \begin{pmatrix} a & b & \# & \# \\ b & c & \# & \# \\ c & a & b & \# \\ d & b & c & \# \\ \underline{a} & \underline{b} & \# & \# \\ \underline{b} & \underline{c} & \# & \# \end{pmatrix}, Z_5 = \begin{pmatrix} a & b & \underline{a} & \underline{d} \\ b & c & \underline{b} & \underline{c} \\ c & a & b & \# \\ d & b & c & \# \\ a & b & \# & \# \\ b & c & \# & \# \end{pmatrix},$$

$$Z_6 = \begin{pmatrix} a & b & a & d \\ b & c & b & c \\ c & a & b & \# \\ d & b & c & \# \\ a & b & \underline{a} & \underline{b} \\ b & c & \underline{b} & \underline{c} \end{pmatrix}, Z_7 = \begin{pmatrix} a & b & a & d \\ b & c & b & c \\ c & a & b & \# \\ d & b & \underline{c} & \underline{c} \\ a & b & \underline{d} & \underline{b} \\ b & c & b & c \end{pmatrix}, Z_8 = \begin{pmatrix} a & b & a & d \\ b & c & b & c \\ c & a & \underline{b} & \underline{b} \\ d & b & \underline{c} & \underline{c} \\ a & b & d & b \\ b & c & b & c \end{pmatrix}.$$

The occurrence of the possibly occluded objects in X are underlined. From this construction, it is obvious that the generating sequence of a partially occluded image may not be unique. The decomposition of X into objects is not unique due to the fact that some objects may be partially or totally occluded by others. For example, since S_2 and S_3 share identical first rows, if the second column of S_2 or S_3 is occluded in the image X then there is no way to differentiate between the two objects.

From the above example we can see that there exists many possible generating sequences for a given image, since it's decomposition is not unique. In fact, it can be shown that the number of distinct generating sequences may be exponential in the size of the input image, see [IS-97]. This fact complicates the design of an iterative algorithm for decomposing or even validating a two-dimensional image since it is imperative not to inspect all possible generating sequences for a given image. The definition of a valid image implies trivially that the objects are contained within the image X. That is, we assume that there is no S_i for all $i \in \{1, \ldots, k\}$ that is "cut off" on the edges of the image.

To analyse in more detail a partially occluded image we need to extend the notion of a prefix in two dimensions, which is not as clear and well defined as in the one dimensional case.

Definition 2.2 Let X be an array of size $r \times s$. Then we define a *row-prefix* of X as a rectangular sub-array of X occurring at positions $X[i_1..i_2, 0..j]$ for some $0 \leq i_1 \leq i_2 \leq r-1$ and $0 \leq j \leq s-1$. If $i_1 = 0$ then the sub-array $X[0..i_2, 0..j]$ is called a *proper row-prefix* of X. Similarly for columns, let a *column-prefix* of X be a rectangular sub-array of X occurring at $X[0..i, j_1..j_2]$ for some $0 \leq i \leq r-1$ and

$0 \leq j_1 \leq j_2 \leq s-1$. If $j_1 = 0$ then we say that the sub-array is a *proper column-prefix* of X. See Figure 1(i) for an example of a row-prefix and a column-prefix of X.

We can define a *row-suffix* and a *column-suffix* in a similar way.

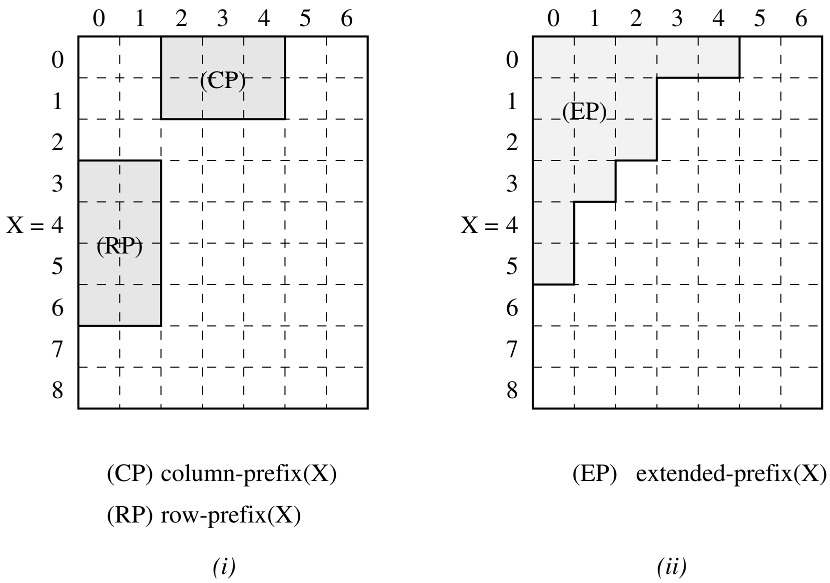


Figure 1: (i) The rectangular sub-array $X[3..6, 0..1]$ is a row-prefix of X and $X[0..1, 2..4]$ is a column-prefix of X. (ii) The staircase like array of points $\mathcal{P} = \{X[0..5, 0..\ell_i - 1] : 0 \leq i \leq 5, \ell_i = [5,3,3,2,1,1]\}$ is a decreasing extended-prefix of X since $\ell_i \leq \ell_{i-1}\ \forall\ 0 \leq i \leq 5$.

This leads to four basic resulting facts if X is a valid image over the set of objects $\mathcal{O} = \{S_1, \ldots, S_k\}$:

Fact 1: For some $i \in \{1..k\}$ there exists a *prefix* of S_i that is also a prefix of X.

Fact 2: For some $i \in \{1..k\}$ there exists a *row-prefix* and a *column-prefix* of S_i occurring at positions $X[l_1..l_2, 0..j]$ and $X[0..l, j_1..j_2]$ respectively with $0 \leq l_1, l_2, l \leq r-1$ and $0 \leq, j_1, j_2, j \leq s-1$.

Fact 3: For some $i \in \{1..k\}$ there exists a *suffix* of S_i that is also a suffix of X.

Fact 4: For some $i \in \{1..k\}$ there exists a *row-suffix* and a *column-suffix* of S_i occurring at positions $X[l_1..l_2, s-j-1..s-1]$ and $X[r-l-1..r-1, j_1..j_2]$ respectively with $0 \leq l_1, l_2, l \leq r-1$ and $0 \leq j_1, j_2, j \leq s-1$.

This follows from the fact that some of the S_i's must occur on the four edges, the top left hand corner and bottom right hand corner of the image X for it to be valid. In an analogue extension from the paper on one-dimensional occluded strings [IS-97], we now break down the validity of a given image into three families of representations of a valid image.

Proposition 2.1 *Let X be an $r \times s$ array over Σ which contains no background symbols $\#$'s. Let $\mathcal{O} = \{S_1, \ldots, S_k\}$ be a set of objects all being $m \times m$ square arrays.*

The array X is a valid image *over $\mathcal{O}$ if and only if one of the following conditions holds:*

$$X = \left(\begin{array}{c|c} \begin{array}{c}\text{proper}\\ \text{pref}(S_i)\end{array} & Y_1 \\ \hline Y_2 & Y_3 \end{array}\right) \tag{7}$$

$$X = \left(\begin{array}{c|c} Y_1 & Y_2 \\ \hline Y_3 & \begin{array}{c}\text{proper}\\ \text{suff}(S_i)\end{array} \end{array}\right) \tag{8}$$

$$X = \left(\begin{array}{c|c|c} Y_1 & Y_2 & Y_3 \\ \hline Y_4 & \text{sub}(S_i) & Y_5 \\ \hline Y_6 & Y_7 & Y_8 \end{array}\right) \tag{9}$$

where the following applies for each equation:

The image resulting from the superposition of an $m \times m$ array of symbols $\#$ on top of $properpref(S_i)$, $propersuff(S_i)$ or $sub(S_i)$ together with the resulting sub-arrays Y_j, $j \in \{1,2,3\}$ for equation (7) and (8) and Y_j, $j \in \{1,\ldots,8\}$ for equation (9) must be valid images over $\mathcal{O}$. □

By using the above classification on valid images together with Facts 1 to 4, we aim to achieve a method for efficiently detecting invalid images as a primary task in the design of the algorithm. However before doing so, we need to refine further the notion of a prefix and a suffix of a two-dimensional array.

Definition 2.3 Let X be an array of size $r \times s$. Then we denote by an *extended-prefix* or *staircase prefix* of X a subset of points of X such that:

$$\mathcal{P} = \{X[0..r', 0..\ell_i - 1] : 0 \leq i \leq r' \leq r-1, 0 \leq \ell_i \leq s\}$$

where ℓ_i is either an increasing or decreasing monotone sequence. If $\ell_i \leq \ell_{i-1}$ $\forall i \in \{0..r'\}$ then $\mathcal{P}$ is a decreasing extended-prefix and an increasing extended-prefix otherwise. See Figure 1 (ii) for an example of an decreasing extended-prefix of X. If $r' < r-1$ and $\ell_i < s$, $\forall i \in \{0..r'\}$ then we say that $\mathcal{P}$ is a *proper* extended-prefix of X

We can define an *extended-suffix* of X in a symmetrical way.

Following the decomposition that was achieved in Proposition 2.1, we define the validity of a partially occluded image using extended-prefixes.

Proposition 2.2 *Let X be an $r \times s$ image over Σ which contains no background symbols $\#$'s. Let $\mathcal{O} = \{S_1, \ldots, S_k\}$ be a set of $m \times m$ square arrays called the objects. Let $\mathcal{P}(S_j) = \{S[0..m', 0..\ell_i - 1] : 0 \leq i \leq m' \leq m-1, 0 \leq \ell_i \leq m\}$ be a decreasing extended-prefix of some object $S_j \in \mathcal{O}$ occurring at position $X[p,q]$. Then the image X is* valid *over the set of objects $\mathcal{O}$ if and only if the following occurs for any extended-prefix $\mathcal{P}(S_j)$ of X.*

Lets assume first that $\mathcal{P}(S_j)$ is a proper extended-prefix of S_j. Then we claim that every neighbour to the right of the perimeter of the extended-prefix is the occurrence of a row-prefix *or a* column-prefix *of some object in $\mathcal{O}$ for the image X to be valid. If $\mathcal{P}(S_j)$ is a non-proper extended-prefix of S_j then we can only claim that the neighbouring point must be a member of a prefix, a suffix or a substring of some object in $\mathcal{O}$ for the image X to be valid.* □

A similar breakdown can be achieved for increasing extended-prefixes and both increasing and decreasing extended-suffixes.

3 Data Structures and Main Techniques

The algorithm presented here checks the validity of a given partially occluded image according to the definition of a valid image in Definition 2.1. The aim of the algorithm is to decompose the occluded image into a finite set of obstructed objects. If X is valid over the set of objects $\mathcal{O}$ then the algorithm returns a (possible) generating sequence for X, as described in Example 2.1. One can easily extend the case of a rectangular image to that of a square image. In the following, we decompose a square image with a set of square objects. Let X be a $n \times n$ array of symbols drawn from some alphabet Σ called the image. Let a dictionary $\mathcal{O} = \{S_1, \ldots, S_k\}$ consist of a finite set of distinct objects representing $m \times m$ arrays of symbols drawn from Σ.

The main methods used in the algorithm rely upon the computation of the occurrences in X of the longest extended-prefixes and extended-suffixes of the objects in $\mathcal{O}$ and in particular chains of longest extended-prefixes and extended-suffixes in X. In order to achieve this we need to maintain and update several data structures.

Each extended-prefix occurring in X has an associated *prefix-head* at the start of each row and an associated *prefix-tail* at the end of each row. During the iterations of the algorithm we aim to concatenate overlapping extended-prefixes resulting in *extended-prefix-chains*. Every chain is given a head for the first extended-prefix and a tail for the last extended-prefix at every row. All data structures mentioned above also apply in a symmetrical way to deal with suffixes. These data structures will be described in detail in the full version of this paper.

So as to perform queries concerning the longest prefixes of each row of any object occurring in the image X we need to use a trie representing common prefixes.

Definition 3.1 The *common prefix tree* of k strings r_i of length m is a rooted trie (digital search tree) with k leaves such that:

1. Each edge of the tree is labelled with a symbol from the alphabet Σ and is directed away from the root.

2. No two edges emanating from the same node have the same label.

3. Each leaf u is uniquely identified with a string r_i, in the sense that the concatenation of the labels on the path from the root to u is r_i.

4. Each internal node v of height $1 \leq h \leq m-1$ in the tree represents a subset of strings having a common prefix of length h.

3.1 Preprocessing

Step 1:

Construct the Aho-Corasick [AC-75] automata for the rows of each objects in the dictionary $\mathcal{O} = \{S_1, \ldots, S_k\}$. Let $r_i^{(j)} = S_j[i, 0..m-1]$ denote the ith row of the jth object in the dictionary. Let $R = \{r_0^{(1)}, \ldots, r_{m-1}^{(1)}, \ldots\ldots, r_0^{(k)}, \ldots, r_{m-1}^{(k)}\}$.

Building the Aho-Corasick automata for R takes $O(km^2 \log |\Sigma|)$ time, since there are m rows each of length m for each of the k objects in the dictionary and the Aho-Corasick depends on the alphabet Σ.

Step 2:

Construct a *common prefix tree* Γ_i for each of the rows of the objects in the dictionary.

Given a fixed row i, we build Γ_i by refining Γ_{i-1}. First, we set up a first path of length m for the ith row of the first object $S_1[i, 0..m-1]$. Now we do a character comparison for each of the remaining rows by walking down the tree that is being built by querying the automata build in Step 1 and branching out when the symbols are not equal. However the procedure of walking down the tree must be prefix conserving according to Γ_{i-1}.

Once the tree is constructed, we order the internal nodes of the tree by assigning indices to each of internal node. Each such index will represent a subset of objects having a common prefix.

Building a common prefix tree for the k objects and a given row takes $O(km \log k)$ time. To build m such trees (i.e for each row) will take $O(km^2 \log k)$ time.

Step 3:

Preprocess the trees build in Step 2 for answering Lowest Common Ancestor (LCA) queries. By using the algorithm by Harel and Tarjan [HT-84] we can perform this type of query in constant time allowing linear time, in the size of the input, for preprocessing. This will help us answer constant time queries in the prefix tree concerning the longest prefixes of objects.

Step 4:

Create a linked list from the final states in the Aho-Corasick automata of Step 1 pointing to the index of the node they belong to in the associated common prefix tree.

Step 5:

Build a $n \times n$ table $START$, which stores the occurrence of the longest prefix of the *first* row among the objects for each position in X. Initialize a bulletin board of size $n \times n$ corresponding to each position in X. Each position in the bulletin board $START$ stores the following values:

$$START[i,j] = (\ell, \delta), \quad 0 \leq \ell \leq m-1, \quad 1 \leq \delta \leq k,$$

where ℓ represents the length of the *longest prefix* of the *first row* of any object in $\mathcal{O}$. The *unique* identifier δ represents the index found in the prefix tree corresponding to a subset of objects that share a common prefix of length ℓ in their first row. This table of size n^2 can be computed by using the common prefix tree for the first row (Γ_0) of the objects in $O(mn^2 \log k)$ time.

The values computed in the table $START$ identify the starting position of an extended-prefix. After these preprocessing steps we claim that the following can be achieved during the computation of the main algorithm:

Corollary 3.1 *Given the Aho-Corasick automaton $AC(R)$ computed above for all the km rows of the objects the query of testing whether*

$$\text{prefix}_l(X[i, j'..j'+m]) = \text{prefix}_l(r_i^{(j)})$$

requires constant time for a fixed row i.

This fact will allow us to perform $O(m)$ constant time queries for each position in X, yielding a total time complexity of $O(mn^2)$ in the main algorithm.

Step 6:

Do all previous steps (1–5) for dealing in a symmetrical way with suffixes. That is, we reverse all the rows of the objects and compute the AC automata for these rows.

Computing the longest common row prefix trees and preprocessing these for answering LCA queries is straightforward. Then we need to add to the $START$ table the additional suffix values (ℓ, δ), for the longest prefixes of the last row of the objects.

3.2 Main Algorithm and Sub-procedures

The main ideas of the algorithm are outlined below. The algorithm iterates over the points of X by sweeping from left to right over the rows of X. During the iterations we will use the Aho-Corasick automaton and the prefix trees from the preprocessing to answer queries concerning the occurrence of longest row prefixes and suffixes of objects appearing in X. Additionally, we will use a data structure called a "window" of size $2m \times n$ which stores the information of extended-prefixes and extended-suffixes of objects occurring for each position in X. The process of building a decreasing extended-prefix at an arbitrary point $X[p,q]$ with (ℓ, δ) from $START[p,q]$ is done in the following way:

procedure build_extended_prefix$(\ell, \delta, X[p, q..q+m-1])$
begin
$(\ell_0, \delta_0) := (\ell, \delta)$;

$i := 1$;
while $\ell_{i-1} \neq 0$ **do**
  $\ell_i :=$ length of longest prefix of the ith row ;
  $\delta_i :=$ index of node at height ℓ_i in prefix tree ;
  Comment: feed $X[p, q..q+m-1]$ to the prefix tree Γ_i.
  if $\delta_i \neq \delta_{i-1}$ **then** $(\ell_i, \delta_i) \leftarrow LCA(\delta_{i-1}, \delta_i)$;
  else $\ell_i := min\{\ell_{i-1}, \ell_i\}$;
  $W[i] := \ell_i$;
  $i := i + 1$;
return array W and the final δ_i
end

One can extend this construction to the one of building increasing extended-prefixes and extended-suffixes in a symmetrical way.

The aim of this sweeping technique is to create extended-prefixes and extended-suffixes and chain them together to create valid sub-images. For each point that needs to be validated we use the following *decomposition principles* which are based on Proposition 2.2:

(i) The occurrence of an extended-prefix of an object in a valid image must be followed by a (not necessarily proper) row/column prefix of an object.

(ii) If an occurrence of a extended-prefix of an object in an image is followed by an occurrence of a proper extended-suffix of an object, then the image is not valid. In a valid image, the occurrence of a proper extended-suffix of an object is always preceded by the extended-suffix of an object.

(iii) The occurrence of a extended-suffix of an object in a valid image can be followed by either a prefix, a suffix or a proper substring of an object.

(iv) The occurrence of a sub-array of an object in a valid image is preceded and followed by valid images.

3.2.1 Step 0: Initialization of data structures

Initialization: Validate position $X[0, 0]$
begin
Initialize a $2m \times n$ array called the "window" W.
(1) **validate current row** $X[0, 0..m-1]$.
  if $\ell \in START[0, 0]$ **then** $(\ell, \delta) \leftarrow START[0, 0]$;
    mark $W[0, 0..\ell - 1]$ with $X[0, 0]$ as *prefix-chain-head*
  else return 'image not valid'. Stop.

(2) **Validate next row** $X[1, 0..\ell - 1]$.
  $(\ell', \delta') \leftarrow START[1, 0]$;
  if $\ell' \in START[1, 0]$ **then** Stop.
    Comment: start of a new extended-prefix on next iteration ;
  else expand the extended-prefix starting at $X[0, 0]$.
    Comment: looking recursively for an extended-prefix down the rows

at most $m-1$ times using procedure build-extended-prefix.

end

3.2.2 Main algorithm

The main steps of the algorithm are as follows:

1. Building chains of extended-prefixes and extended-suffixes using the procedures for augmenting them row by row and the decomposition principles.
2. Creating valid sub-images by concatenating adjacent extended-prefix-chains and extended-suffix-chains.
3. Two-dimensional pattern matching on the remaining blocks of substrings.

The full details of the algorithm will appear in the forthcoming full version of this paper.

4 Conclusion and open problems

The algorithm presented here can be extended to handle variable length objects. An interesting open practical problem is the validation of images with sets of objects that are concave or non-continuous; of particular interest is the variant of the problem with objects over $\Sigma \cup \{\Lambda\}$, where Λ is a transparent symbol and this alphabet defines a set of strings with holes. Another interesting problem is the computation of the *depth* of an object in an image, i.e. the number of objects applied onto an object after the placement of an object in an image. Finally, approximate occlusion analysis is of practical importance and therefore all the above mentioned problems need to be extended to handle errors.

References

[AFI-91] A. Apostolico, M. Farach and C.S. Iliopoulos, Optimal superprimitivity testing for strings, *Information Processing Letters*, (1991), 39, 17–20.

[AC-75] A.V. Aho and M.J. Corasick, Efficient string matching: an aid to bibliographic search, *Comm. ACM*, (1975), 18(6), 333–340.

[BC-94] W. Bischof and T. Caelli, Learning structural descriptions of patterns: a new technique for conditional clustering and rule generation, *Pattern Recognition*, (1994), 27(5), 689–699.

[CIK-98] M. Crochemore, C.S. Iliopoulos and M. Korda, Two-dimensional prefix string matching and covering on square matrices, *Algorithmica*, (1998), 20, 353–372.

[DS-96] A.M. Duval and W.F. Smyth, Covering a circular string with substrings of fixed length, *Int. J. of Foundations of Computer Science*, (1996), 7(1), 87–93.

[HT-84] D. Harel and R.E. Tarjan, Fast algorithms for finding nearest common ancestors, *SIAM J. Comput.*, (1984), 13(2), 338–355.

[IR-97] C.S. Iliopoulos and J.F. Reid, An optimal parallel algorithm for analysing occluded images, In *Proc. 4th Annual Australasian Conference on Parallel And Real-Time Systems*, (1997), University of Newcastle, Australia. N. Sharda and A. Tam (eds), Springer-Verlag, 104–113.

[IS-97] C.S. Iliopoulos and J. Simpson, On-line validation and analysis of occluded images, In *Proc. 8th Australasian Workshop on Combinatorial Algorithms*, (1997), Research on Combinatorial Algorithms, Queensland University of Technology, Australia, V. Estivill-Castro (ed), 25–36.

[KMP-77] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM J. Comput.*, (1997), 6, 322-350.

[MS-94] D.W.G. Moore and W.F. Smyth, An optimal algorithm to compute all the covers of a string, *Inform. Process. Lett.*, (1994), 50(5), 239–246.

[MS-95] D.W.G. Moore and W.F. Smyth, A correction to: An optimal algorithm to compute all the covers of a string, *Inform. Process. Lett.*, (1995), 54, 101–103.

[PL-94] P.A. Pevzner and R.J. Lipshutz, Towards DNA sequencing chips, In *Proc. 19th Int. Symp. on Mathematical Foundations of Computer Science*, (1994), Lecture Notes in Computer Science, Springer-Verlag, 841, 143–158.

Application of Sequence Alignment Methods to Multiple Structural Alignment and Superposition[1]

Arthur M. Lesk

Department of Haematology
University of Cambridge Clinical School
MRC Centre
Hills Road
Cambridge CB2 2QH
United Kingdom

e-mail: `aml2@mrc-lmb.cam.ac.uk`

Abstract. With the goal of developing efficient multiple structural alignment methods, we have asked which of the pairwise structure alignment methods lends itself most readily to generalization to multiple structure alignment. A simple linear encoding of the sequence and associated residue conformation can be treated by standard multiple *sequence* alignment methods.

Key words: Protein structure, multiple alignment

1 Introduction

One often wishes to analyse proteins that have similar folding patterns but too little sequence similarity to permit the alignment of their residues by sequence-based methods. Such proteins may be very distant relatives, or independently-evolved examples of the same folding pattern. For only two structures, it is possible to perform a structural alignment; that is, to identify residues that occupy similar spatial positions within the structure [GL98]. However, just as multiple sequence alignments are far more informative than pairs of aligned sequences, so the analysis of protein structures requires alignment of more than two sequences.

Most previous approaches to multiple structure alignment have been based on pairwise structural alignments. The simplest approach is to choose a master structure and align all the others to it. This has the obvious limitations of dependence on the choice of the master structure, and failure to make use of relationships between pairs of non-master sequences. Lesk & Fordham [LF96], in a study of the chymotrypsin-like serine proteases, did structural alignments of all pairs of structures, and collated the results into a common alignment table. However, the experience with those calculations suggests that it would be useful to ask whether any of the known pairwise structural

[1]This work was supported by the Wellcome Trust.

superposition methods lends itself to generalisation to a true multiple superposition approach. The problem is only to determine the residue–residue correspondences, that is, the alignment. Once the alignment is known methods are available for the multiple superposition of the molecules [SBPL92],[D92].

There have been numerous approaches to the problem of structural superposition (for a review see [GL98]). Some operate in three-dimensional space, and are based on detection of small well-fitting pieces and combining them [VS91],[ATG92]; others are based on similarity of contact matrices [HS93],[NRTZ95],[L95].

However, the methods that would seem to be most directly generalizable to multiple alignment are those that reduce the three-dimensional structural superposition problem to a one-dimensional problem. There are several ways to achieve this. One is to characterise each residue by its pattern of neighbours [LVW85],[TO89]. Another is to characterize each residue by its mainchain conformation [LSW84],[KdHN89]. (It is clear that these approaches depend on the linear nature of the polypeptide chain.) Still another is to classify each position in a polypeptide chain by its environment; this also has application to structure prediction by asking whether a particular sequence is compatible with a succession of encoded environments [BLE91].

In this report we pursue the idea that after encoding a protein by a one-dimension characterization of the successive residues, together with limited amino acid sequence information, multiple sequence alignment methods can be applied to produce a multiple structure alignment. We use a set of distantly-related globins as an example and test of feasibility of the method.

Other approaches to multiple structure alignment have been published by Russell & Barton [RB92], Taylor, Flores & Orengo [TFO94], and May & Johnson [MJ95]. Our approach is similar to that of Šali & Blundell [ŠB90].

2 Co-ordinates and Calculations

All co-ordinates are taken from the Protein Data Bank [B77]. For multiple sequence alignment we used the program map, by Huang [H94].

We assign to each residue a symbol that combines information from the amino acid sequence and from the residue conformation.

2.1 Encoding the sequence: reduced amino acid alphabet

We encode the amino acid sequence according to a reduced alphabet corresponding to physico-chemical classes of amino acids:

Table 1. Reduced alphabet based on classifying amino acids into types of similar physicochemical properties

GAST	small nonpolar
CVILP	small/medium hydrophobic
FYMW	large hydrophobic
NQH	polar
DE	charged, negative
KR	charged, positive

2.2 Encoding the conformation

We make use of Efimov's dissection of the Sasisekhan–Ramachandran diagram [E93], with modifications: The conformation of the mainchain of a protein is specified by conformational angles ϕ, ψ and ω. Values of ω are limited to narrow ranges around $+180°$ and $-180°$. Allowed ranges for ψ and ϕ are limited by steric constraints to discrete regions which can be charted in the Sasisekharan–Ramachandran plot. We use the nomenclature of Efimov [E93] but extend his regions to assign to each residue a symbol for the region to which it is closest. (Efimov's definitions cover only a subset of the possible values of ϕ and ψ.) In this way we encode the structure of a protein as a sequence of conformation states of the individual residues:

Table 2. Classification of mainchain conformations based on that of A.V. Efimov [E93]

A	α_r — right-handed $\alpha-$helix
B	β — extended strand
D	throat between α and β regions
L	α_l — left-handed $\alpha-$helix
E	bottom of $+/-$ region (in which $\phi > 0, \psi < 0$)
C	cis-peptide
X	other

From the previous two tables we have assigned to each residue one of six symbols based on its amino acid identity, and one of 7 symbols based on its conformation. By assigning a unique symbol to each possible combination of these we represent each residue by a single character in a 42-character alphabet. Each element of the substitution matrix associated with this alphabet is the sum of a contribution from change in amino acid class (see Table 1) and a contribution from change in conformation class (see Table 2) according to the following rules:

Contribution from amino acid classification:

Same class	uncharged $\leftrightarrow$ uncharged (including polar)	uncharged $\leftrightarrow$ charged
10	5	0

Contribution from conformational classification:

Same class	different class
0	-10

The initiate-gap penalty was 20 and the extend-gap penalty 5.

3 Results

We have implemented the methods described and applied them to three distantly-related globin structures: sperm whale myoglobin, bloodworm globin and leghaemoglobin from yellow lupin. The results are as follows. (The symbols, which correspond to the assignment of a unique character to each ordered pair of reduced amino acid alphabet and residue conformation, should be considered arbitrary.)

```
                     .    :    .    :    .    :    .    :    .    :    .    :
Sperm whale       HBYAYMSGGGSGMA4GYAOG--AASASYGGG4GM4AUGYAGY4NY4M4S----H4BYAY
Bloodworm         HBAAS4SGGAAAM4YGAECOVDAAGA44GGG4MGAAUGSMAAGMDNA------EACZGA
Yellow lupin     EJBYASAAGG4AAMYYMSAUG--G4SAS4MMGGGGYGCGAA4YGNAMG5EBAZHHSUTGY

                     .    :    .    :    .    :    .    :    .    :    .    :
Sperm whale      M4ABYYG44SAGAGGAAGAAGG447EUUYAYG4GGASASA--A4S7HHG4MGYMGAYAGG
Bloodworm        GAAGAA4GGASGAGAGAUGAOYA4MG---ASM4AGAG4S4CNES5TH5ASMMYGGAAAGG
Yellow lupin     GSA-SAA4GM4GGMYAAGSGYGAEHHHBZAAG4SGAAGSG--A6-DHBYASMGGG4YAGG

                     .    :    .    :    .    :    .    :    .    :    .    :
Sperm whale      SGGSA4UGAYOBAYASAAMS4AGYGM44YGAA4M4YGDNV
Bloodworm        AAMYS4GEA4OTAAA4YAMAAAMAYGAAAGGAAGS
Yellow lupin     4AG4YGGEA6NBYYGSAAMAGAMYYGAGGG44YMYYA
```

A translation of these results back to the amino acid sequence follows:

```
Sperm whale      VLSEGEWQLVLHVWAKVEADV--AGHGQDILIRLFKSHPETLEKFDRFKH----LKTEAE
Bloodworm        GLSAAQRQVIAATWKDIAGADNGAGVGKKCLIKFLSAHPQMAAVFGFS------GASDPG
Yellow lupin    GALTESQAALVKSSWEEFNANI--PKHTHRFFILVLEIAPAAKDLFSFLKGTSEVPQNNPE

                     .    :    .    :    .    :    .    :    .    :    .    :
Sperm whale      MKASEDLKKHGVTVLTALGAILKKKGHHEAELKPLAQSHA--TKHKIPIKYLEFISEAII
Bloodworm        VAALGAKVLAQIGVAVSHLGDEGKMV---AQMKAVGVRHKGYGNKHIKAQYFEPLGASLL
Yellow lupin     LQA-HAGKVFKLVYEAAIQLEVTGVVVTDATLKNLGSVHV--SK-GVADAHFPVVKEAIL

                     .    :    .    :    .    :    .    :    .    :    .    :
Sperm whale      HVLHSRHPGDFGADAQGAMNKALELFRKDIAAKYKELGYQG
Bloodworm        SAMEHRIGGKMNAAAKDAWAAAYADISGALISGLQS
Yellow lupin     KTIKEVVGAKWSEELNSAWTIAYDELAIVIKKEMDDAA
```

In contrast, the following results are from applying the same multiple sequence alignment program to the sequences alone:

```
Sperm whale       VLSEGEWQLVLHVWAKVE-ADV-AGHGQDILIRLFKSHPETLEKFDRFKHLKTEAEMKA
Bloodworm         GLSAAQRQVIAATWKDIAGADNGAGVGKKCLIKFLSAHPQMAAVFG-FS--------GA
Yellow lupin     GALTESQAALVKSSWEEFN-ANI-PKHTHRFFILVLEIAPAAKDLFS-F--LKGTSEVPQ

                     .    :    .    :    .    :    .    :    .    :    .    :
Sperm whale      SE-DLKKHGVTVLTALG-AI--LKKKGHHEAE--LKPLAQSH---ATKHKIPIKYLEFIS
Bloodworm        SDPGVAALGAKVLAQIGVAVSHLGDEGKMVAQ--MKAVGVRHKGYGNKH-IKAQYFEPLG
Yellow lupin     NNPELQAHAGKVFKLVYEAAIQLEVTGVVVTDATLKNLGSVHVSKG----VADAHFPVVK

                     .    :    .    :    .    :    .    :    .    :    .    :
Sperm whale      EAIIHVLHSRHPGDFGADAQGAMNKALELFRKDIAAKYKELGYQG
Bloodworm        ASLLSAMEHRIGGKMNAAAKDAW-----------AAAYADIS--GALISGLQS
Yellow lupin     EAILKTIKEVVGAKWSEELNSAW-----------TIAYDEL----AIV--IKKEMDDAA
```

The results were checked against the published structural alignments [LC80],[BCL87], and it can be stated that the structure-based calculation performed somewhat better than the purely sequence-based one. However, extensive tests on a variety of systems are required to evaluate the effectiveness of the method properly. We suggest that the results presented here encourage further development of the approach.

Conclusions

We have designed and implemented a simple method for multiple structural alignment, using a one-dimensional representation of the conformation of a polypeptide chain, combined with the sequence, and standard multiple *sequence* alignment methods to perform the alignment.

References

[ATG92] Alexandrov, N.N., Takahashi, K. & Gō, N. (1992). Common spatial arrangement of backbone fragments in homologous and non-homologous proteins. J. Mol. Biol. 225. 5–9.

[BCL87] Bashford, D., Chothia, C. & Lesk, A.M. (1987). Determinants of a protein fold: Unique features of the globin amino acid sequences J. Mol. Biol. 196, 199–216.

[B77] Bernstein, F.C, Koetzle, T.F., Williams, G.J.B., Meyer, E.F. Jr., Brice, M.D., Rodgers, J.R., Kennard, O., Shimanouchi, T. & Tasumi, M. (1977). The Protein Data Bank: a computer based archival file for macromolecular structures. J. Mol. Biol. 112, 535–542.

[BLE91] Bowie, J.U., Lüthy, R. & Eisenberg, D. (1991). A method to identify protein sequences that fold into a known three-dimensional structure. Science 253: 164–170.

[D92] Diamond, R. (1992). On the multiple simultaneous superposition of molecular structures by rigid body transformations. Protein Science 1, 1279–1287.

[E93] Efimov, A.V. (1993). Standard structures in proteins. Prog. Biophys. Molec. Biol. 60, 201-239.

[GL98] Gerstein, M. & Levitt, M. (1998) Comprehensive assessment of automatic structural alignment against a manual standard, the scop classification of proteins. Prot. Sci. 7, 1–12.

[HS93] Holm, L. & Sander, C. (1993). Protein structure comparison by alignment of distance matrices. J. Mol. Biol. 233, 123–138.

[H94] Huang, X. (1994) On global sequence alignment. Computer Applications in the Biosciences 10, 227–235.

[KdHN89] Karpen, M.E., de Haseth, P.L. & Neet, K.E. (1989). Comparing short protein substructures by a method based on backbone torsion angles. Proteins: Structure, Function and Genetics 6, 155–167.

[LC80] Lesk, A.M. & Chothia, C. (1980). How different amino acid sequences determine similar protein structures: The structure and evolutionary dynamics of the globins J. Mol. Biol. 136, 225–270.

[L95] Lesk, A.M. (1995). Three-dimensional pattern matching in protein structure analysis In: Combinatorial Pattern Matching, Z. Galil, E. Ukkonen, eds. Lecture Notes in Computer Science 937. Springer-Verlag, Berlin, pp. 248–260.

[LF96] Lesk, A.M. & Fordham, W.D. (1996). Conservation and variability in the structures of serine proteases. J. Mol. Biol. 258, 501–537 (1996).

[LSW84] Levine, M., Stuart, D. & Williams, J. (1984). A method for systematic comparison of the three-dimensional structures of proteins and some results. Acta crystallographica A40, 600–610.

[LVW85] Liebman, M. N., Venanzi, C.A. & Weinstein, H. (1985). Structural analysis of carboxypeptidase A and its complexes with inhibitors as a basis for modelling enzyme recognition and specificity. Biopolymers 24, 1721–1758.

[MJ95] May, A.C.W. & Johnson, M.S. (1995). Improved genetic algorithm-based protein-structure comparisons – pairwise and multiple superpositions. Protein Engineering 8, 873–882.

[NRTZ95] Nichols, W.L, Rose, G.D, Ten Eyck, L.F. & Zimm, B.H. (1995). Rigid Domains in Proteins: An Algorithmic Approach to their Identification. Proteins: Structure, Function, Genetics 23, 38–48.

[RB92] Russell, R. B. & Barton, G. J. (1992), Multiple sequence alignment from tertiary structure comparison: assignment of global and residue confidence levels. PROTEINS: Struc. Func. Genet., 14, 309–323.

[ŠB90] Šali, A & Blundell, T.L. (1990). Definition of general topological equivalence in protein structures. A procedure involving comparison of properties and relationships through simulated annealing and dynamic programming. J. Mol. Biol., 212, 203–228.

[SBPL92] Shapiro, A., Botha, J.D., Pastore, A. & Lesk, A.M. (1992). A method for multiple superposition of structures. Acta Crystallographica A48, 11–14.

[TFO94] Taylor, W.R., Flores, T.P. & Orengo, C. (1994) Multiple protein structure alignment. Prot. Sci. 3, 1858-1870.

[TO89] Taylor, W.R. & Orengo, C.A. (1989). Protein structure alignment. J. Mol. Biol. 208, 1–22.

[VS91] Vriend, G. & Sander, C. (1991). Detection of common three-dimensional substructures in proteins. Proteins: Structure, Function and Genetics 11, 52–58.

Approximate String Matching by Fuzzy Automata

Václav Snášel

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

e-mail: `Vaclav.Snasel@uplo.cz`

Abstract. I explain new ways of construction of search algorithms using fuzzy sets. I define Fuzzy Automata and I discuss the possibilities of use.

Key words: fuzzy automata, searching

The Factor Automaton[1]

Milan Šimánek

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech technical University
Karlovo nám. 13, 121 35 Prague 2

e-mail: `simanek@fel.cvut.cz`

Abstract. The direct acyclic word graph (DAWG) is a good data structure representing a set of strings related to some word with very small space complexity. The famoust DAWG is the factor DAWG which is representing the set Fac(*text*) of all factors (substrings) of the string *text*. Bellow we call factor DAWG as DAWG. Finite automaton implementing this data structure is able to make out any substring of string *text* in time proportional only to length of the substring while its space complexity is linear to the length of the string *text*. We can define several operations on DAWG. Operations are usefull for fast derivating of the DAWG automaton from a similar one. This paper concern operation L-delete on factor graph DAWG and the relationship between deterministic and nondeterministic factor automaton.

Key words: DAWG, factor automaton, substring, pattern matching, fast searching

1 Introduction

The factor automaton is a finite automaton which accepts the set of all substrings of the string [1, chapter 6]. The set of all substrings (factors) of the string *text* is Fac(*text*).

This factor automaton can be formulated as a deterministic one or a nondeterministic one. The nondeterministic factor automaton is a good abstraction for formal description of its behaviour and of operations performed on it. On the other hand the deterministic one is used for implementation and practical use. This version is sometimes called direct acyclic word graph, $DAWG$, because it has no transition loop.

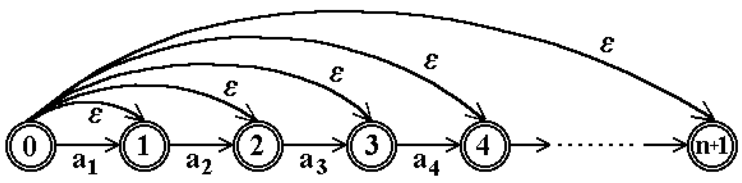
The main advantage of the $DAWG$ is very fast substrings searching while it keeps small memory requirments. Any matching string can be found in time equal to the length of the pattern looking for. The size of the factor automaton $DAWG(text)$ is linear with respect to the length of the string *text*. Total number of the nodes is less then double length of the input string *text*. The proof is in [1, Theorem 6.1].

[1]This research has been supported by GAČR grant No. 201/98/1155

2 Construction

2.1 Nondeterministic factor automaton

The nondeterministic factor automaton, which accepts all substrings of the string *text*, has $N+1$ states and $2N-1$ transitions, where N is the length of the string *text*. The structure of this automaton for string *text*$= a_1 a_2 a_3 a_4 ... a_N$ is shown on the next picture.



2.2 Deterministic factor automaton

The deterministic factor automaton DAWG can be obtained from nondeterministic one [3] or we can construct it step by step using an incremental construction algorithm [1, 6.3 On-line construction]. Although we have a construction algorithm, in general, we cannot say anything about the structure of transitions except nonexistence of the circle and an estimate bounds of the number of states. The pattern matching using this automaton has optimal speed. The number of comparations (or other elementar operations) is linear to the length of the searching pattern.

2.3 Relation between deterministic and nondeterministic automata

It appears that every construction method produces equivalent (isomorphic) deterministic factor automaton. We can say the deterministic factor automaton is the best simulation of the nondeterministic one. In this simulation every state in deterministic automaton corresponds to a set of active states in nondeterministic automaton. This relationship can by very usefull for discovering and proving new algorithms for deterministic automata.

3 Operations on factor automaton

We can define a number of operations on factor automaton. Each operation modify given factor automaton representing string *text* to a new factor automaton representing another string *text'* while strings *text* and *text'* are very similar. It is important that both new and old factor automaton will be similar too and therefore performing the operation spend a little amount of time.

We will deal with this operations on a factor automaton:

operation	**action** *text'*
Append	the string *text'* will be longer by a character
Insert	inserts a character before the first character of the string *text'*
$R-delete$	*text'* is the string *text* without the last character
$L-delete$	*text'* is the string *text* without the first character
Replace	replaces one character in string *text* by another one

The algorithms for some operations have been yet discovered (*Append*, $R{-}delete$), but the algorithms of *Insert* and *Replace* are not known. This article concern about the algorithm of the $L-delete$ operation.

This operation modifies DAWG($a_0a_1a_2a_3...a_n$) to another factor automaton accepting all substrings of the string $a_1a_2a_3...a_n$ which is by a first character a_0 shorter then the original string $a_0a_1a_2a_3...a_n$. The algorithm is shown bellow.

The combination of operations *Append* and *L-delete* enables fast searching in the compression method known as LZ77 which use so called sliding window. Sliding window contains a part of source text with constatnt length. The window is moving through the text so at the begining it contains the first k characters of the text and at the end operating it contains the last k characters of the source text.

4 DAWG in details

To enable incremental construction of this factor automata (*append* operations) requires to keep a bit more information about the DAWG working on. In every step we should know the set of states (a state of finite automaton per a node of the DAWG), transitions between the states (representing edges of the DAWG), and the fail function. The fail function is used for creating and extending DAWG. We will need know which is the next character for each state for the *L-delete* operation.

Before we will show the algorithm we should make some denotation. *Next(q)* is a following character in source string *text* for each state q in the DAWG factor automaton. Concatenation of $Next(q_0) + Next(Next(q_0)) + ...$ gives the string *text* for DAWG(*text*). There is defined the fail function *Fail(q)* for each state q of DAWG automaton. If the automaton is in the state q_1 after reading substring uv and in the state q_2 after reading substring v which is the longest possible then $Fail(q_1){=}q_2$. Factor automaton being in state q_2 accepts each suffix which is accepted in state q_1. $Skip(q)$ is the set of states p_i which $Fail(p_i)$ is equal to state q. Function $Skip$ is the inverse function of function $Fail$: $p \in Skip(q)$ iff $q = Fail(p)$.

5 The algorithm of L-delete operation

Let main chain is a sequence of states q_0, $q_1{=}\delta(q_0, \mathrm{Next}(q_0))$, ..., $q_i{=}\delta(q_{i-1}, \mathrm{Next}(q_{i-1})$, ..., q_n. The idea of this algorithm is to disable passing only through a part of the main chain but to protect passing anyway through at least one skip transition.

This algorithm duplicates the starting part of main chain of states. One copy (original) of begin of main chain is used for processing these substrings which will pass through some skip transition later. Second copy (duplicated) is used for processing these states which have passed some skip transition before.

Not all main chains will be duplicated. The duplication process stops at the state where is obvious which shift transition will be pass. This stop state is determined by a value of Skip function. Assume last duplicated state is r. Next state to be duplicate is s. Let state $t = \delta(s, Next(s))$ is the next state after s. If the set of states Skip(t) is empty then duplication process stops, because no shift transition can be pass. If the set of states Skip(t) contain only one state, then duplication process stops too, because only one shift transition is possible and therefore it can be done immediately. Otherway if the number of states Skip(t) is greater then one then duplication process continue.

INPUT:	DAWG(aw)
OUTPUT:	DAWG(w)
LOCAL VARIABLES:	**a** – a character
	q_0, q_1, **r**, **s**, **t**, **d** – states
	q_0 – the initial state

```
a:=Next(q0)
q1:=δ(q0,a)
if |Skip(q1)| = 0 then
       δ(q0,a) := nil
       delete(q1)                                    possible recursive delete
else if |Skip(q1)| = 1 then
       δ(q0,a) := Skip(q1)
       delete(q1)                                    possible recursive delete
else
       r:=q0
       s:=q1
       loop
              a:=Next(s)
              t:=δ(s,a)
              if |Skip(t)| < 2 then break
              d:=duplicate(t)
              δ(r,a):=d
              Fail(t):=d
              r:=d
              s:=t
       endloop
       if |Skip(t)| = 1 then
              δ(s,a):=Skip(t)
       else
              δ(s,a):=nil
       endif
endif
```

6 Time and memory complexity

It seems that the time complexity of one *L-delete* operation is at least constant or in the worst case linear to length of the text *text*. The DAWG(*text*) for string *text* of length N has at most $2 * N$ states [1]. Therefore the time complexity of sequence of N *L-delete* operations is linear to N.

The number of states of DAWG(*text*) is limited by $2.N$ where N is number of characters in source string *text*. Moreover, DAWG(*text*) has less than $3.N$ edges. This is independent of the size of the alphabet [1, Theorem 6.1].

7 Conclusion

The power of operation *L-delete* grows up in conjunction with the operation *append*. We can apply k-times operation append which constructs the base DAWG for first k characters of the text. Then we will apply repeatively a couple of operations *L-delete* and *append*. We will get a moving window for fast searching in this part of the text. The speed of searching is independent of size of the searching window and depends only on the size of pattern looking for. The main application can be LZ77 compression algorithm. The part consuming the largest amount of time is just the algorithm searching for a pattern in a searching window. Using this searching algorithm should speed up compression.

References

[1] Crochemore, M., Rytter, W.: **Text Algorithms,** chapter 6, Subword graphs, Oxford University Press, 1994

[2] Chen, M. T., Seiferas, J.: **Efficient and elegant subword tree construction,** Combinatorial Algorithms on Words, NATO Advanced Science Institutes, Series F, vol. 12, Springer-Verlag, Berlin, 1985, 97–107

[3] Melichar, B.: **The construction of factor automaton**, Workshop 98, Czech Technical University, Prague 1998, 189–190

Directed Acyclic Subsequence Graph[1]

Zdeněk Troníček and Bořivoj Melichar

Department of Computer Science and Engineering
Czech Technical University
Karlovo náměstí 13, 121 35 Prague 2, Czech Republic
phone: ++420 2 2435 7287, fax: ++420 2 298098

e-mail: {`tronicek,melichar`}`@fel.cvut.cz`

Abstract. Directed Acyclic Subsequence Graph is an automaton, which accepts all subsequences of the given string. We introduce a left-to-right algorithm for incremental construction of DASG. The algorithm requires $\mathcal{O}(z)$ extra space and $\mathcal{O}(nz \log z)$ time for arbitrary alphabet ($\mathcal{O}(nz)$ for fixed alphabet), where $z = min(|\Sigma|, n)$. The number of transitions can be reduced by encoding input symbols using k digits, where $k < min(|\Sigma|, n)$. We introduce a left-to-right algorithm for incremental construction of DASG for $k = 2$. We show the extension of the algorithm for the set of strings and its application for the longest common subsequence problem.

Key words: Directed Acyclic Subsequence Graph, finite automaton, searching subsequences

1 Introduction

A subsequence of a string is any string obtained by deleteing zero or more symbols from the given string. Directed Acyclic Subsequenc Graph (DASG) is an automaton, which accepts all subsequences of the given text. It was introduced in [2] (preliminary version was published in [1]). DASG is analogous to Directed Acyclic Word Graph (DAWG) [3] using subsequences instead of substrings.

Let us suppose an alphabet Σ and a text $T = t_1 t_2 \dots t_n$ over this alphabet. DASG for the text T is an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an input alphabet, δ is a transition function, q_0 is the initial state and F is a set of final states. States are denoted by numbers in this article.

In [2], there is described a right-to-left algorithm for construction of DASG and encoding for reducing the number of transitions. In section 3 we introduce an incremental left-to-right algorithm for construction of DASG, and in section 4 its modification for encoded DASG. In section 5 we show the extension of the algorithm for a set of strings and its application for the longest common subsequence problem.

[1]This research has been supported by GAČR grant No. 201/98/1155

2 Motivation

Let $Sub(T)$ denotes the set of all subsequences of the text $T = t_1t_2 \dots t_n$. The set $Sub(T)$ can be described recursively by the regular expression (ε is empty subsequence):
$Sub_0 = \varepsilon$
$Sub_i = Sub_{i-1}(\varepsilon + t_i)$
$Sub(T) = Sub_n$
For the set Sub_n then holds:

$$Sub_n = Sub_{n-1}(\varepsilon + t_n) = Sub_{n-2}(\varepsilon + t_{n-1})(\varepsilon + t_n) = (\varepsilon + t_1)(\varepsilon + t_2) \dots (\varepsilon + t_n)$$

This expression gives us the direction for construction of the nondeterministic version of DASG. The example of such nondeterministic finite automaton (NFA) is in Fig. 1. It also holds:

$$Sub_n = Sub_{n-1} + Sub_{n-1}t_n = \varepsilon + Sub_0t_1 + Sub_1t_2 + \dots + Sub_{n-2}t_{n-1} + Sub_{n-1}t_n$$

The last expression can be used for construction of the nondeterministic DASG without ε-transitions (the example is in Fig. 2). If all the symbols of T are different, we obtain directly the deterministic finite automaton (DFA). The example of deterministic DASG is in Fig. 3.

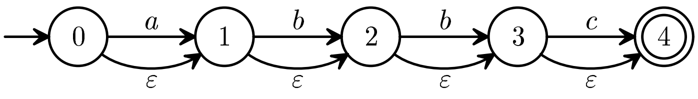


Figure 1: NFA accepting all subsequences of $T = abbc$ (with ε-transitions).

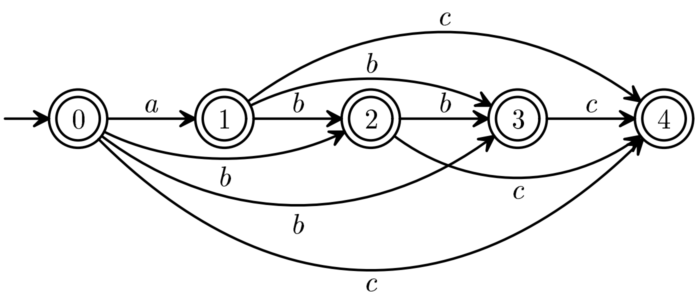


Figure 2: NFA accepting all subsequences of $T = abbc$ (without ε-transitions).

3 Construction of DASG

Let us suppose an alphabet Σ and a text $T = t_1t_2 \dots t_n$ over this alphabet. For each symbol a of the alphabet Σ we will maintain the value f_a, which is the smallest number of the state not having an output transition labeled with a. We start with an automaton with the only state 0. Each state of the automaton is final.

Lemma 1: The automaton constructed by Algorithm 1 has $(n + 1)$ states.

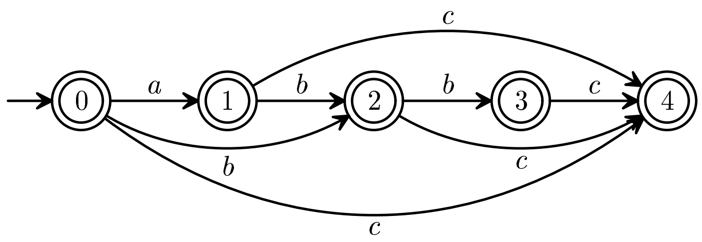


Figure 3: DASG for the string $T = abbc$.

```
for each a ∈ Σ do
    f_a ← 0
end for
for k ← 1 to n do
    add state k
    for s ← f_{t_k} to (k − 1) do
        add a transition labeled t_k from the state s to the state k
    end for
    f_{t_k} ← k
end for
```

Figure 4: Algorithm 1 (incremental construction of DASG)

Proof: We start with the automaton with one state. The main cycle of the algorithm is performed n times. During each step of the cycle we add just one new state.

Lemma 2: The automaton constructed by Algorithm 1 accepts just all subsequences of T.

Proof: We prove the lemma in two steps.
1. If S is a subsequence of the string T then S is accepted by the automaton (induction by the length of S):
Step 1: $|S| = 1, S = s_1$. If s_1 occurs in T then state 0 of the automaton has transition labeled with s_1 and the automaton accepts S.
Step 2: A string $S_k = s_1 s_2 \ldots s_k$ is a subsequence of T and is accepted by the automaton. Let us create a new string $S_{k+1} = s_1 s_2 \ldots s_k s_{k+1}$ by adding a symbol s_{k+1} to the end of S_k. There exists a sequence $i_1, i_2, \ldots, i_k$ such that $s_1 = t_{i_1}, s_2 = t_{i_2}, \ldots, s_k = t_{i_k}$ (the automaton will finish in state i_k after accepting S_k). If there exists i_{k+1} such that $i_k < i_{k+1} \leq n$ and $s_{k+1} = t_{i_{k+1}}$, then state i_k has transition labeled with s_{k+1} and the automaton accepts S_{k+1}.
2. If S is accepted by the automaton then S is a subsequence of T (induction by the length of S):
Step 1: $|S| = 1, S = s_1$. If S is accepted by the automaton then state 0 has the transition labeled with s_1. State 0 has transition labeled with s_1 only if there exists $j, 1 \leq j \leq n$ such that $s_1 = t_j$.
Step 2: A string $S_k = s_1 s_2 \ldots s_k$ is accepted by the automaton and there exists a sequence $i_1, i_2, \ldots, i_k$ such that $s_1 = t_{i_1}, s_2 = t_{i_2}, \ldots, s_k = t_{i_k}$. We create a new string $S_{k+1} = s_1 s_2 \ldots s_k s_{k+1}$ by adding a symbol s_{k+1} to the end of S_k. The automaton will finish in state i_k after accepting S_k. If state i_k has transition labeled s_{k+1} then there exists $i_{k+1}, i_k < i_{k+1} \leq n$ such that $s_{k+1} = t_{i_{k+1}}$. $\triangle$

3.1 Number of transitions

Definition 1: Let Σ be an alphabet and $T = t_1 t_2 \dots t_n$ a string over this alphabet. Let Σ_e denotes the set of all symbols, which are contained in T. We define the effective size of Σ as $z = |\Sigma_e|$.

The minimum number of transitions is n (if and only if all the symbols of T are the same).
For each state k, the maximum number of its output transitions is:

$$max\_out\_deg_k = min(z, n-k)$$

It results from that the first $(n+1-z)$ states may have at most z output transitions and for the last z states the maximum number of output transitions decreases to 0. Then, the maximum total number of transitions is:

$$(n+1-z)z + (z-1) + \ldots + 2 + 1 + 0 = (n+1-z)z + \frac{z(z-1)}{2} = \frac{2zn + z - z^2}{2}$$

DASG has this number of transitions if and only if the last z symbols of T are different.

3.2 Complexity

The main cycle of the Algorithm 1 (lines 4–10) is performed n times. The lines 5 and 9 take constant time. At lines 6–8 are added all the transitions. Therefore, the total time complexity of lines 6–8 is $\mathcal{O}(nz)$.

For fixed alphabet we suppose that adding or looking up the transition takes constant time. Then, the algorithm requires time $\mathcal{O}(n + nz)$ in the worst case. The time of subsequence test is $\mathcal{O}(m)$ in the worst case.

For arbitrary alphabet we suppose that adding or looking up the transition takes time $\mathcal{O}(\log z)$. Then, all the complexities must be multiplied by factor $\log z$. In this case, we use a balanced tree for values $f_a, a \in \Sigma$.

In both cases the algorithm requires $\mathcal{O}(z)$ extra space.

4 Encoding of input symbols

Encoding as a method for reducing the number of transitions was introduced in [2]. The method use $k < z$ digits for encoding the input symbols, where z is the effective size of the alphabet. The number of states grows at most to $n\lceil \log_k z \rceil + 1$, but the number of transitions usually decreases (see [2] for details). Fig. 5 shows the encoded version of DASG for the text $T = abbc$ (in this case encoding does not reduce the number of transitions). The symbols are coded this way: $a = 00$, $b = 01$, $c = 10$.

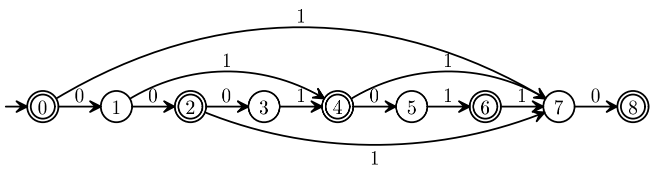


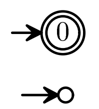
Figure 5: Encoded DASG for the string $T = abbc$.

Let us suppose an alphabet Σ and a text $T = t_1t_2 \dots t_n$ over this alphabet. Each symbol a of Σ we encode using digits 0 and 1. For that we need at least $c = \lceil \log_2 z \rceil$ digits. The algorithm is incremental. When we add a new symbol encoded as $e_1e_2 \dots e_c$, we need to ensure, that all previous final states have an output path labeled with $e_1e_2 \dots e_c$. We use a binary tree as an auxiliary structure. The tree is built during the construction of the automaton. Each inner node of the tree has two lists (for 0 and for 1), which contents states, where ends the path labeled with the same symbols as the path in the tree, starting at any final state and is the longest possible. So, if a state s is in the list l_e in the node with the path $p_1p_2 \dots p_q$ from the root, there exists a path in the DASG from any final state to state s, which is labeled $p_1p_2 \dots p_q$ and state s has no output transition labeled e. We start with an automaton with the only state 0. States (tc) for $t = 0, 1, \dots, n$ are final. At the beginning, the tree has only the initial node.

$l_0^\varepsilon \leftarrow \{0\}$, $l_1^\varepsilon \leftarrow \{0\}$
for $k \leftarrow 0$ to $(n-1)$ **do**
    encode the symbol $t_{(k+1)}$ as $e_1e_2 \dots e_c$
    set the root as the actual node in the tree
    for $b \leftarrow 1$ to c **do**
        add state $(ck + b)$
        for each state s in the list l_{e_b} in the actual node of the tree **do**
            add a transition labeled e_b between states s and $(ck + b)$
            remove s from the list l_{e_b}
        end for
        go to the child of the actual node of the tree through the transition labeled e_b and set the child as the actual node (if the child does not exist, create it and set both lists of new node empty)
        if $b < c$ **then**
            add the state $(ck + b)$ to the both lists in the actual node
        end if
    end for
    mark the state $(ck + c)$ as a final state and add it to the both lists in the root of the tree
end for

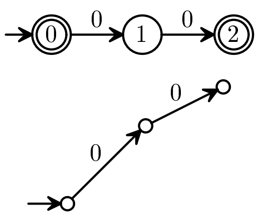
Figure 6: Algorithm 2 (incremental construction of encoded DASG)

The algorithm is demonstrated in Fig. 7–11. The lists maintained in the node with the path p from the root are denoted as l_0^p and l_1^p, the symbols are coded this way: $a = 00$, $b = 01$, $c = 10$.



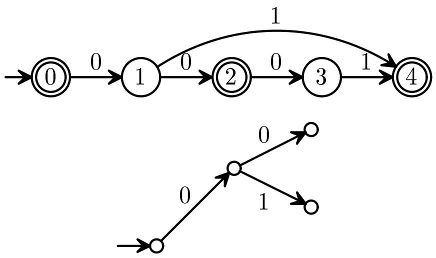
$l_0^\varepsilon = \{0\}$, $l_1^\varepsilon = \{0\}$

Figure 7: Encoded DASG for the string $T = \varepsilon$.



$l_0^\varepsilon = \{2\}$, $l_1^\varepsilon = \{0, 2\}$
$l_0^0 = \{\}$, $l_1^0 = \{1\}$

Figure 8: Encoded DASG for the string $T = a$.



$l_0^\varepsilon = \{4\}$, $l_1^\varepsilon = \{0, 2, 4\}$
$l_0^0 = \{3\}$, $l_1^0 = \{\}$

Figure 9: Encoded DASG for the string $T = ab$.

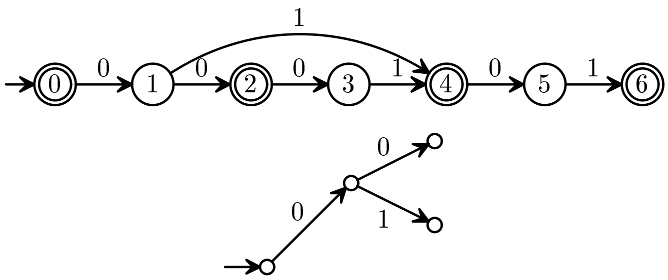
4.1 Number of transitions

The number of states grows to $1 + nc = 1 + n\lceil \log_2 n \rceil$. The maximum number of transitions is $c(2n - \frac{1}{2}(c+1)) = \lceil \log_2 z \rceil (2n - \frac{1}{2}(\lceil \log_2 z \rceil + 1))$.

4.2 Complexity

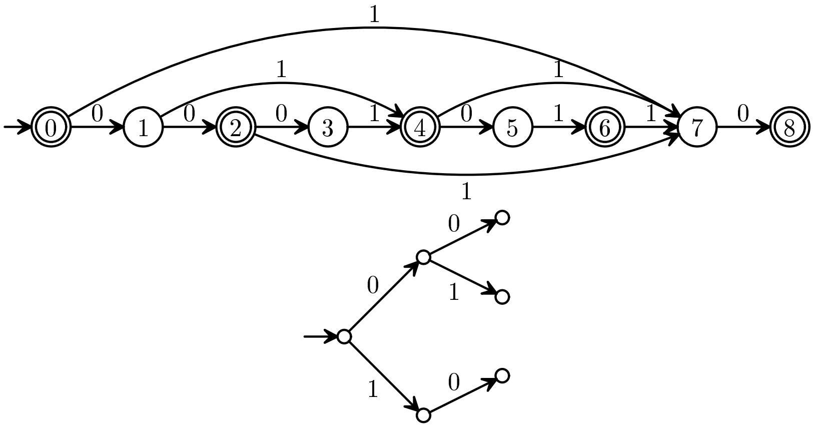
The main cycle (line 2–15) is performed n times. Lines 1,3,4,6,11,12,13 and 16 require constant time. The cycle on line 5 is performed $\mathcal{O}(\log_2 z)$ times. The total number of transitions is $\mathcal{O}(n \log_2 z)$. Therefore, the total time complexity of lines 7–10 is $\mathcal{O}(n \log_2 z)$. Hence, the total time complexity is $\mathcal{O}(n \log_2 z)$.

The algorithm needs $\mathcal{O}(z + n \log_2 z)$ extra space for the tree and for the lists in its nodes. The subsequence test requires $\mathcal{O}(m \log_2 z)$ time in the worst case.



$l_0^\varepsilon = \{6\}$, $l_1^\varepsilon = \{0, 2, 4, 6\}$
$l_0^0 = \{3, 5\}$, $l_1^0 = \{\}$

Figure 10: Encoded DASG for the string $T = abb$.



$l_0^{\varepsilon} = \{6, 8\}$, $l_1^{\varepsilon} = \{8\}$
$l_0^0 = \{3, 5\}$, $l_1^0 = \{\}$
$l_0^1 = \{\}$, $l_1^1 = \{7\}$

Figure 11: Encoded DASG for the string $T = abbc$.

5 DASG for a set of strings

Let us suppose an alphabet Σ and strings $T_1, T_2, \ldots, T_w$ over this alphabet. We extend Algorithm 1 to a set strings $\{T_1, T_2, \ldots, T_w\}$. Let $L = \sum_{i=1}^{w} length(T_i)$.

The construction of DASG has two steps:

- Construction of inverted trie for reversed strings.
- Construction of an automaton.

Construction of inverted trie: Inverted trie arises from trie by reversing the transitions and can be constructed during the construction of trie (each node of trie will have one inverted transition). Final nodes of trie are initial nodes of inverted trie. Inverted trie is used as an auxiliary structure and served for finding common suffixes of the strings.

Construction of an automaton: For each string $T_i, 1 \leq i \leq w$ we will maintain:

- lists $l_a^i, a \in \Sigma$ of states, which have no output transition for the symbol a
- actual position act_i in inverted trie
- actual position $last_i$ in the automaton

In each node of inverted trie we save the number of corresponding state in the automaton. Let v denotes this number, let γ denotes the transition function in trie, and let δ denotes the transition function in the automaton. For each transition of the automaton we have to remember, which strings it belongs to. This set is denoted E. We start with the automaton with the only state 0. Each state of the automaton is final. Set is ordered set with two defined operation: $first$ return the first string in the set and $next$ return the successor of the string. The total number of states after each step is in the variable $states$.

The algorithm is demonstrated in Fig. 15–18.

```
for i ← 1 to w do
   for each a ∈ Σ do
      l^i_a ← {0}
   end for
   act_i ← final state for the string T_i in trie
   last_i ← 0
end for
for each node i in trie do
   v(i) ← 0
end for
Set ← {T_1, T_2, ..., T_w}
states ← 1
c ← 1
p ← 1
for k ← 1 to L do
   M ← ∅
   symbol ← p-th symbol of T_c
   act_c ← γ(act_c, symbol)
   if δ(last_c, symbol) ≠ ∅ then
      new_state ← δ(last_c, symbol)
   else if v(act_c) > 0 then
      new_state ← v(act_c)
   else
      add state states
      new_state ← states
      v(act_c) ← states
      states ← states + 1
   end if
   last_c ← new_state
   for each s ∈ l^c_symbol do
      if δ(s, symbol) ≠ ∅ then
         M ← M ∪ {δ(s, symbol)}
         E(s, symbol) ← E(s, symbol) ∪ {c}
      else
         δ(s, symbol) ← new_state
         E(s, symbol) ← {c}
      end if
   end for
   l^c_symbol ← ∅
   M ← M ∪ {new_state}
   for each a ∈ A do
      l^c_a ← l^c_a ∪ M
   end for
   if p = length(T_c) then
      Set ← Set \ {T_c}
   end if
   if next(Set, c) is defined then
      d is defined as follows: next(Set, c) = T_d
   else
      d is defined as follows: first(Set, c) = T_d
      p ← p + 1
   end if
   c ← d
end for
```

Figure 12: Algorithm 3 (extension of DASG for a set of strings $\{T_1, T_2, \ldots, T_w\}$)

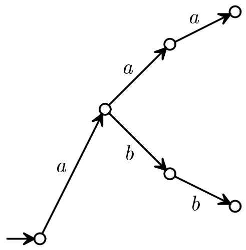


Figure 13: Trie for the reversed strings aaa and bba.

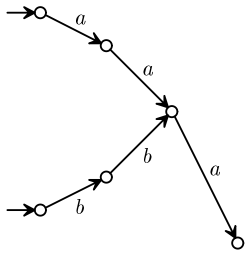
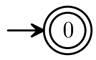
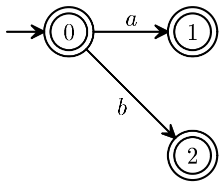


Figure 14: Inverted trie for the reversed strings aaa and bba.



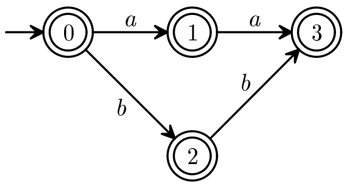
$l_a^1 = \{0\}$, $l_b^1 = \{0\}$
$l_a^2 = \{0\}$, $l_b^2 = \{0\}$

Figure 15: Extension of DASG for the $Set = \{\varepsilon\}$.



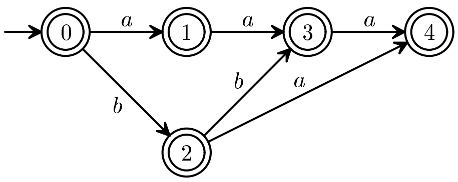
$l_a^1 = \{1\}$, $l_b^1 = \{0, 1\}$
$l_a^2 = \{0, 2\}$, $l_b^2 = \{2\}$
$E(0, a) = \{1\}$, $E(0, b) = \{2\}$

Figure 16: Extension of DASG for the $Set = \{a, b\}$.



$l_a^1 = \{3\}$, $l_b^1 = \{0, 1, 3\}$
$l_a^2 = \{0, 2, 3\}$, $l_b^2 = \{3\}$
$E(0, a) = \{1\}$, $E(0, b) = \{2\}$
$E(1, a) = \{1\}$, $E(2, b) = \{2\}$

Figure 17: Extension of DASG for the $Set = \{aa, bb\}$.



$l_a^1 = \{4\}$, $l_b^1 = \{0, 1, 3, 4\}$
$l_a^2 = \{1, 4\}$, $l_b^2 = \{1, 3, 4\}$
$E(0, a) = \{1\}$, $E(0, b) = \{2\}$
$E(1, a) = \{1\}$, $E(2, b) = \{2\}$
$E(3, a) = \{1\}$, $E(2, a) = \{2\}$

Figure 18: Extension of DASG for the $Set = \{aaa, bba\}$.

5.1 Number of states

For each symbol of the string, except for the last, a new state can be added. Hence, the DASG constructed in Algorithm 3 has at most $1+\sum_{i=1}^{w}(length(T_i)-1)+1 = 2+L-w$ states. DASG has this number of states if no two strings have any common nonempty prefix and suffix.

Each state can have at most z output transitions. Therefore, the total number of transitions is $\mathcal{O}(Lz)$.

5.2 Complexity

Construction of inverted trie requires $\mathcal{O}(L)$ time and $\mathcal{O}(L)$ extra space. For the time analysis of the Algorithm 3 is important the time complexity of set operations. We use four of them: *insert a member*, *delete a member*, *assign a value* and *union*. Suppose, that we use a balanced tree for the representation of the set M (another possibilities are a member function or a list). Then, the operations *assign* and *union* require $\mathcal{O}(|M|)$ time and the other operations require $\mathcal{O}(\log |M|)$ time.

Lines 1–7 require $\mathcal{O}(wz)$ time, lines 8–10 require $\mathcal{O}(L)$ time, line 11 requires $\mathcal{O}(w)$ time. The main cycle (lines 15–53) is performed L times. Line 16 requires $\mathcal{O}(L)$ time, and lines 18–20 require $\mathcal{O}(\log z)$ time. The cycle on the lines 30–38 is performed at most L times. Its time complexity is $\mathcal{O}(L \log L)$ (line 32 requires $\mathcal{O}(\log L)$ time). Line 40 requires $\mathcal{O}(\log L)$ time, lines 41–43 require $\mathcal{O}(Lz)$ time. Hence, the total time complexity is $\mathcal{O}(L^2 + L^2 \log L + L \log L + L^2 z) \approx \mathcal{O}(L^2 \log L)$ for arbitrary alphabet.

We need $\mathcal{O}(L)$ space for trie, $\mathcal{O}(L)$ space for the set M, and $\mathcal{O}(Lwz)$ space for the lists l_a^c. Hence, the total required extra space is $\mathcal{O}(Lwz)$.

5.3 Application: the longest common subsequence problem

The longest common subsequence (LCS) problem is known problem with applications in many areas. There are efficient algorithms that solve the LCS problem for two strings, for example [4].

Let us define the following problem (as in [2]): What is the longest common subsequence between any $k \leq w$ strings in a set S of w strings?

To solve this problem, we construct DASG for the set S and append to each transition $\delta(q, a)$ the number of strings in the set E (denoted as $num(q, a)$) and to each state q the number of its input edges (denoted as c_q) and the number of its input edges with $num(q, a)$ greater or equal k (denoted as ck_q). We do not need the set E in this case. Then, we traverse DASG. During the traversing we use LIFO (Last-In-First-Out) memory as an auxiliary structure (denoted as $Stack$). Dot (.) denotes concatenation. The longest sequence of input symbols from the initial state to the state q is stored in cs_q.

```
lcs ← ε
for each state q do
   cs_q ← ε
end for
Stack ← 0
while Stack is not empty do
   q ← Pop
   if length(cs_q) > length(lcs) then
      lcs ← cs_q
   end if
   for each a ∈ Σ such that δ(q, a) ≠ ∅ do
      r ← δ(q, a)
      c_r ← c_r − 1
      if c_r = 0 then
         Push(r)
      end if
      if num(q, a) ≥ k then
         ck_r ← ck_r − 1
         if ck_r = 0 then
            cs_r ← cs_q.a
         end if
      end if
   end for
end while
```

Figure 19: Algorithm 4 (the longest common subsequence)

The traversion of DASG requires $\mathcal{O}(Lz)$ time. For common subsequences cs_q we need $\mathcal{O}(Ly)$ space, where $y = max\{length(T_i)\}$. Hence, the general longest common subsequence problem of w strings requires $\mathcal{O}(L^2 \log L + Lz)$ time for arbitrary alphabet. It is a better solution than presented in [2].

6 Conclusion

In section 3, we introduced a left-to-right algorithm for construction of DASG. It requires $\mathcal{O}(nz \log z)$ time and $\mathcal{O}(z)$ extra space for arbitrary alphabet. The subsequence test takes $\mathcal{O}(m \log z)$ time.

In section 4, we showed the modification of that algorithm for encoded DASG. The modified algorithm requires $\mathcal{O}(n \log z)$ time and $\mathcal{O}(z + n \log z)$ extra space. The subsequence test takes $\mathcal{O}(m \log z)$ time.

In section 5, we extended DASG for a set of strings and used it to solve the general longest common subsequence problem. Construction of DASG takes $\mathcal{O}(L^2 \log L)$ time and $\mathcal{O}(Lwz)$ extra space. The traversion of DASG requires $\mathcal{O}(Lz)$ time. Hence, the solution of the general longest common subsequence problem requires $\mathcal{O}(L^2 \log L)$ total time and $\mathcal{O}(Lwz + Ly)$ space, where $y = max\{length(T_i)\}$.

References

[1] Baeza-Yates, R. A.: The Subsequence Graph of a Text. TAPSOFT'89, Proceedings of the International Joint Conference on Theory and Practice of Sotware Development, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming (CAAP'89), Lecture Notes in Computer Science 351, Barcelona, Spain, March 1989, pages 104–118.

[2] Baeza-Yates, R. A.: Searching subsequences. Theoretical Computer Science 78 (1991), pages 363–378.

[3] Crochemore, M., Rytter, W.: Text algorithms. Oxford University Press, 1994.

[4] Hunt, J., Szymanski, T.: A fast algorithm for computing longest common subsequences. Communication of ACM 20, 1977, pages 350–353.

[5] Hirschberg, D. S.: A linear space algorithm for computing maximal common subsequences. Communication of ACM, 18(6), 1975, pages 341–343.

An Early-Retirement Plan for the States

Bruce W. Watson[1,2] and Richard E. Watson[2]

[1]Department of Computer Science
University of Pretoria
Hillcrest 0083, Pretoria
South Africa

[2]Ribbit Software Systems Inc.
IST Technologies Research Group
Box 24040, 297 Bernard Ave., Kelowna
British Columbia, V1Y 9P9, Canada

e-mail: watson@cs.up.ac.za, {watson, rwatson}@RibbitSoft.com

Abstract. New applications of finite automata, such as NLP and asynchronous circuit simulation, can require automata of millions or even billions of states. All known construction methods (in particular, the interesting reachability-based ones that save memory, such as the subset construction, and simultaneously minimizing constructions, such as Brzozowski's) have intermediate memory usage much larger than the final automaton, thereby restricting the maximum size of the automata which can be built. In this paper, we present a reachability-based optimization which can be used in any one of the construction algorithms to reduce the intermediate memory requirements. The optimization is presented in conjunction with an easily understood (and implemented) canonical automaton construction algorithm.

Key words: finite automata, very large automata, automata construction, memory constraints, reachability algorithms

1 Introduction

Automata (in the form of acceptors or transducers) are now being heavily used in computational linguistics applications, hardware simulation, text indexing and searching applications. In contrast to their traditional use in compilers, these newer applications make use of automata that are several orders of magnitude larger (in terms of both states and transitions, and therefore memory consumption) than previously contemplated. This can lead to memory problems with constructing the automata and also to runtime inefficiencies[1].

Automata can be constructed in a number of ways, however, in this paper we restrict ourselves to building them from regular expressions (REs). Constructing an automaton from an RE proceeds (conceptually) in three phases:

[1]Addressing some of the runtime inefficiencies is the subject of [4].

1. An abstract state automaton is built, in which each state contains additional information (it is an *object* in memory), for example a position within the regular expression. This additional information allows us to determine the out-transitions from the state and whether the state is final or not. The additional information is typically encoded as one of the following:
   - a set of *items* (dots) representing positions in the RE (this is used in all of the item-set constructions [3] and these are used in this paper);
   - a set of *symbol-positions* representing the specific symbols (within the RE) which can be seen next (this is used in the Berry-Sethi, McNaughton-Yamada-Glushkov and Aho-Sethi-Ullman constructions [3]); or
   - an RE representing a *derivative* of the original RE (this is used in Antimirov's and Brzozowski's constructions [3]).
2. From the abstract state automaton, we build a concrete automaton (isomorphic to the abstract state automaton) in which each state is represented only as an integer, with a single bit devoted to indicating whether it is final or not.
3. The abstract state automaton is *retired*, freeing up the memory, leaving only the concrete one.

Of course, a real implementation would not directly implement this conceptual model. As we see in §2, the construction of the final automaton representation could be done incrementally as parts of the underlying abstract state automaton are built. Still, all of the abstract states are kept in memory until the whole of the final automaton is built, after which they are freed. In the case of a deterministic automaton of a million states, the concrete representation may require less than 32MB. Unfortunately, each abstract state can take up a lot more memory as its concrete counterpart, so the intermediate data-structures could have a peak memory requirement of up to 2GB. Clearly, this is beyond even the realistic *virtual* memory capacity of an average processor and operating system. In this paper, we present Ribbit's solution to the problem.

All of the abstract states are usually kept in memory throughout the construction process since a transition (from the state under construction) can go to any one of the other abstract states. In the optimization, we use a reachability relation to determine which abstract states are no longer reachable during the construction phase. Those abstract states may then be removed from memory (retired).

This optimization technique is quite different from the obvious (and well-understood) optimization of removing unreachable states — which yields smaller automata. In our later discussion, that optimization happens to be included (simply by our use of reachability during automata construction), but the new optimization goes much further — reducing the memory used during the construction process.

2 A canonical automata construction method

In this section, we briefly outline a canonical[2] construction method for deterministic automata. The algorithm is essentially a reachability-based version of the traditional

[2]It is a *canonical* construction because it is used as the starting point of a taxonomy in [3].

three-step algorithm outlined in §1. Since the construction will only be used to illustrate the retirement plan, we will not present it formally. See [3] for a more in-depth discussion of various construction algorithms.

In the construction method, each abstract state consists of a set of *items* where an item is a dot[3] placed within the input RE (in much the same way as the LR parsing item appears within grammar production right-hand sides). A relation, called 'dot closure', takes an item set and propagates each item through the RE without jumping over alphabet symbols within the RE. More precisely, it is the reflexive and transitive closure of the following relation:

1. A dot before the empty string (ϵ) RE yields the dot after the empty string RE. Symbolically, $\bullet\epsilon$ is mapped to $\epsilon\bullet$.

2. A dot before an alternation (union) yields dots in front of each branch of the alternation. Likewise, a dot after either branch of an alternation yields a dot after the entire alternation. Symbolically, $\bullet(E \cup F)$ is mapped to $(\bullet E \cup \bullet F)$, $(E \bullet \cup F)$ is mapped to $(E \cup F)\bullet$ and $(E \cup F\bullet)$ is mapped to $(E \cup F)\bullet$.

3. A dot before a concatenation yields a dot in front of the first operand. A dot after the left part of a concatenation yields a dot in front of the second part. A dot after the second part yields a dot after the entire concatenation. Symbolically, $\bullet(EF)$ is mapped to $(\bullet E)F$, $(E\bullet)F$ is mapped to $E(\bullet F)$, and $E(F\bullet)$ is mapped to $(EF)\bullet$.

4. A dot before a Kleene closure yields a dot after the entire Kleene closure and a dot before the (single) operand of the closure. A dot after the operand of a Kleene closure yields a dot before the same operand and a dot after the entire Kleene closure. Symbolically, $\bullet(E^*)$ is mapped to $((\bullet E)^*)\bullet$ and $(E\bullet)^*$ is mapped to $((\bullet E)^*)\bullet$.

(For simplicity in this paper, we omit the other possible RE operators such as intersection.) In the algorithm, we maintain the invariant that all of our abstract states already have the dot closure operation applied to them.

To compute the destination abstract state of a transition from an abstract state on a symbol a, do the following:

1. For every dot in front of an a in the abstract state, place a dot behind the corresponding a in the destination abstract state. Include no other dots in the destination.

2. Perform the dot closure operation on the destination abstract state.

For example, abstract state

$$\bullet(\bullet a \cup (\bullet a)b)$$

has a transition on a to

$$(a \bullet \cup (a\bullet)(\bullet b))\bullet$$

The closure of the union of the transition relation (over all alphabet symbols) with the dot closure relation constitutes a reachability relation over abstract states. This reachability relation plays an important role in the optimization.

[3]We speak of 'dots' and 'items' interchangeably.

To determine the start abstract state, place the dot before the entire RE and perform the dot closure operation. A state is final if a dot appears after the main RE.

During the construction, we use a bijective data-structure (which we call the *namer*), which maps abstract states to concrete ones. We also use either a queue or a stack of abstract states[4], called the *ready pool*. For the actual construction algorithm, we perform the following steps:

1. Create the start abstract state and the corresponding new concrete state; place the abstract state in the ready pool and use the namer to map the start state to it.

2. While the ready pool is not empty: select the next state (call it the *current state*), remove it from the pool and do the following:

   (a) Lookup the corresponding abstract state in the namer. If it is a final state, mark the current state as final too.

   (b) For each alphabet symbol a such that $\bullet a$ appears as a subitem in the abstract state, do the following:

      i. Construct the destination abstract state for the alphabet symbol, using the transitions explained earlier.

      ii. Check if the destination abstract state is in the namer. If not, create a new concrete state and map the destination to it in the namer, while placing the destination abstract state in the ready pool (so that its out-transitions will eventually be constructed).

      iii. Construct a concrete transition on the alphabet symbol from the current concrete state to the concrete state which the destination is mapped to.

3. Delete the abstract states, the namer and the ready pool.

Clearly, the contents of the namer grow to include all of the abstract states mapped to their corresponding concrete states and none of these pairs are removed until the final step. Since this is the source of the memory problem, in the next section we consider how to retire as many as possible of the abstract states on-the-fly in the second step.

3 An early-retirement plan

Some of the abstract states (appearing in the namer) may be unreachable, regardless of the sequence of transitions, from any of the abstract states still in the ready pool. Indeed, the only abstract states which will be needed in the namer are those reachable (directly or indirectly) from an abstract state whose concrete state is in the ready pool. This follows directly from our use of a reachability algorithm.

[4]Using a queue for the ready pool leads to constructing the automaton transition graph breadth-first, while a stack leads to a depth-first construction. The data-structure could just as easily contain concrete states, since we have a bijection between abstract and concrete states.

Conceptually, our solution is to compute the reachability relation (or some approximation containing it) for abstract states. After we have removed a state from the ready pool and built its out-transitions (in step two), we traverse the namer and purge any abstract states (and corresponding concrete ones) which are no longer reachable from an element of the ready pool.

Our implementation maintains the set of abstract states which are reachable from any of the states in the ready pool. Using the canonical construction, we can maintain this set, R, particularly cheaply: it is the ϵ and letter transition closure of the set of all items present in the ready pool abstract states. For example, if the ready pool contains two states $(\bullet a) \cup b$ and $a \cup (b\bullet)$, we have $R = ((\bullet a \bullet) \cup (b\bullet))\bullet$. An abstract state, q, in the namer is reachable from one in the ready pool if q's constituent items are entirely contained in the set of possibly reachable items, R. In our example, the start state would be $\bullet((\bullet a) \cup (\bullet b))$ (the dot closure of $\bullet(a \cup b)$), which is not wholly contained in R, is therefore unreachable by either of the states in the ready pool and can be removed from the namer.

For efficiency, we implement a set of items by numbering all of the possible item positions within the input RE (there are a finite number of them) and using bit-vectors to represent the sets of items. Consequently, the closure and set containment operations can be performed extremely quickly on most computer hardware using bitwise instructions.

4 Observations and performance

Benchmarking data are still being collected as this paper is submitted. Preliminary data, collected while constructing a number of very large automata, shows a reduction of required working memory by a factor of roughly two. There is also a significant running time penalty of up to a factor of ten for constructing the automata, even discounting the obvious memory paging time.

There are other variants of the reachability algorithm which are being explored. One of the most interesting possibilities is to determine the number of in-transitions to each abstract state. Once the in-transitions of the corresponding concrete state have all been built, the abstract state can immediately be purged from the namer. Unfortunately, it is difficult to efficiently count a state's in-transitions before they have all been built. It is not yet clear whether this approach will improve efficiency.

Conclusions and comments

A number of conclusions can be drawn about the approach presented here:

- This technique serves only to minimize the amount of memory consumed during the construction of the automaton. It does not optimize the running time of the automaton, or even the memory consumed by the final automaton. As such, it is only applicable to the massive automata which occur in applications such as NLP or hardware simulation. In that role, the technique is not only very effective, but it is also the *only* known technique available.

- The technique presented in this paper can significantly slow the construction process by having to evaluate the reachability relation on abstract states. This tradeoff is necessary when constructing very large automata.

- This technique has become necessary because even virtual memory has its limits. The current generation of programmers thinks in terms of a 32-bit address space (4GB), which appears boundless. Not only is the virtual address space not large enough for the construction of some automata using the older algorithms, but most systems do not have 4GB of virtual memory available (due to limited physical swap space).

- Algorithms for minimizing deterministic finite automata have a similar memory constraint. During the minimization process, the set of states are grouped into equivalence classes, which will each represent a new state in the minimized automaton. (The equivalence classes are essentially abstract states.) If the input automaton is already nearing the limits of the available memory, any reasonable representation of the equivalence classes is unlikely to fit within the memory. It appears that some of the same techniques could be applied, using an incremental minimization algorithm such as Watson's [3]. In the case of acyclic automata, this would yield an algorithm similar to the one presented in [2].

- This technique minimizes the number of abstract states present in the mapping from abstract states to concrete states. There are many potential speed optimizations which can be applied, such as minimizing the number of times an abstract state is copied. These possibilities have not yet been explored.

Acknowledgements:

We would like to thank Nanette Saes for proofreading this paper.

References

[1] Aho, A.V., R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1988.

[2] Daciuk, J., B.W. Watson and R.E. Watson. "Incremental Construction of Minimal Acyclic Finite State Automata and Transducers," also submitted to FSMNLP 98.

[3] Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms.* Ph.D dissertation, Eindhoven University of Technology, The Netherlands, 1995.

[4] Watson, B.W. "Practical Optimizations for Automata," Second Annual Workshop on Implementing Automata, London, Canada, 1997. Also available from `www.RibbitSoft.com/research/watson/`.