Research Report DC–2002–03

Proceedings of the Prague Stringology Conference '02

Edited by Miroslav Balík and Milan Šimánek

September 2002

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

Program Committee

Gabriela Andrejková, Jun-ichi Aoe, Maxime Crochemore, Jan Holub, Costas S. Iliopoulos, Thierry Lecroq, Bořivoj Melichar (chair), Bruce W. Watson

Organizing Committee

Miroslav Balík, Martin Bloch, Jan Holub, Vojtěch Kačírek, Milan Šimánek, Zdeněk Troníček, Ladislav Vagner

URL

http://cs.felk.cvut.cz/psc

Proceedings of the Prague Stringology Conference '02
Published by Vydavatelství ČVUT, Zikova 4, 16635 Praha 6, Czech Republic
Edited by Miroslav Balík and Milan Šimánek
Contact: Prague Stringology Club
Katedra počítačů, ČVUT–FEL
Karlovo nám. 13, Praha 2, Czech Republic.
E-mail: psc@cs.felk.cvut.cz   Phone: +420-2-24357470
Printed by Ediční středisko ČVUT, Zikova 4, Praha 6

© Czech Technical University, Prague, Czech Republic, 2002

ISBN 80-01-02616-7

Table of contents

Preface

The Prague Stringology Conference 2002 (PSC'02) was held at the Department of Computer Science and Engineering of the Czech Technical University in Prague, Czech Republic, on September 23–24, 2002. The conference focused on stringology and related topics. Stringology is a discipline concerned with algorithmic processing of strings and sequences.

The papers submitted were reviewed by the programme committee and eleven were selected for presentation at the conference, based on originality and quality. This volume contains these selected papers.

In years 1996–2000 the Prague Stringology Club Workshops (PSCW's) and the Prague Stringology Conference in 2001 preceded this conference. The proceedings of these workshops and the conference had been published by Czech Technical University and are available on WWW pages of PSC. Selected contributions were published in a special issue of the journal Kybernetika.

The Prague Stringology Club (PSC) was founded in 1996 as a research group at the Department of Computer Science and Engineering of the Czech Technical University in Prague. The goal of PSC is to study algorithms on strings and sequences with a special emphasis on the finite automata theory. The first event PSC organized was the workshop PSCW'96 consisting only of invited talks. However, since PSCW'97 the papers are selected based on peer reviews. The aim is not only to present new results in stringology, but also to have people working on these topics meeting in person.

I would like to thank all those who had submitted papers for PSC'02 as well as the reviewers. A special thank goes to all the members of the programme committee, without whose efforts it would not have been possible to put together such a stimulating program of PSC'02. Last, but not least, my thanks go to the members of the organizing committee for ensuring such a smooth running of the conference.

In Hamilton, Ontario, Canada
on August 2002
Jan Holub

A Work-Optimal Parallel Implementation of Lossless Image Compression by Block Matching

Sergio De Agostino

School of Computing
Armstrong Atlantic State University
11935 Abercorn Street
Savannah, Georgia 31419
USA

e-mail: `agos@armstrong.edu`

Abstract. Storer suggested that fast encoders are possible for two-dimensional lossless compression by showing a square greedy matching LZ1 heuristic for bi-level images, which can be implemented by a simple hashing scheme [S96]. In this paper, we show a work-optimal parallel algorithm using a rectangle greedy matching technique requiring $O(\log M \log n)$ time on the PRAM EREW model, where n is the size of the image and M the maximum size of a rectangle.

Key words: lossless image compression, sliding dictionary, parallel algorithm, PRAM EREW

1 Introduction

Textual substitution compression methods (often called "LZ" methods due to the work of Lempel and Ziv [LZ76]) have been designed by Lempel and Ziv [LZ77, ZL78] and Storer and Szymanski [SS82]. These methods parse a string in *phrases* and replace them with *pointers* to copies, called *targets* of the pointers, that are stored in a *dictionary*. The encoded string is a sequence of pointers (some of which may represent single characters). *Static* methods are when the dictionary is known in advance. By contrast, with *dynamic* methods (LZ1 [LZ77] and LZ2 [ZL78]) the dictionary may be constantly changing as the data is processed (see [BCW90, St88] for references).

Storer [S96] and Storer and Helfgott [SH97] generalized the LZ1 method to lossless image compression and suggested that very fast encoders are possible by showing a square greedy matching LZ1 compression heuristic, which can be implemented by a simple hashing scheme and achieves 60 to 70 percent of the compression of JBIG1 on the CCITT bi-level image test set.

With LZ1 text compression, one simply proceeds from left to right making matches in greedy fashion between a substring in the current position and a copy in the part of the string already seen. A key advantage of LZ1 compression is that decoding is always simple and fast. Another advantage is that it is relatively easy to implement. The two key issues for practical implementations are how the encoder searches for matches and how pointers are encoded.

An image has to be scanned in some linear order. In order to achieve a good compression performance, bidimensional matches have to be computed. In [SH97], a square-match encoding algorithm is proposed using a simple hashing scheme directed to bi-level images. A 64K table with one position for each possible 4x4 subarray is the only data structure used. All-zero and all-one squares are handled differently. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic square, a match or raw data. When there is a match, the 4 x 4 subarray in the current position is hashed to yield a pointer to a copy. This pointer is used for the current square greedy match and then replaced in the hash table by a pointer to the current position.

To improve the compression performance, it was also introduced a slower rectangle greedy matching technique requiring $O(M \log M)$ time where M is the size of the match [SH97]. Therefore, $O(n \log M)$ is the best sequential time for an image of size n if we compress by rectangle matching with M the maximum size of a rectangle.

Both heuristics work with an unrestricted window. In [CDG01] a rectangle greedy matching heuristic using a finite window and a bound to the match size was presented. The heuristic is suitable for a fast implementation similar to the one in [S96] and achieves 75 to 90 percent of the compression of JBIG1 on the CCITT bi-level image test set. In this paper, we show a work-optimal PRAM EREW implementation of lossless image compression by block matching requiring $O(\log M \log n)$ time which uses a rectangle greedy matching technique similar to the one in [CDG01]. The parallel heuristic achieves 95 to 97 percent of the compression of the sequential heuristic mentioned above [CDL02]. In section 2, we show how the sequential heuristic works. In section 3, we explain the parallel algorithm. In section 4, conclusions and future work are given.

2 The Rectangle Greedy Matching Technique

The compression heuristic scans an image n x m row by row (*raster* scan) (the greedy matching technique could work with any other scan described in [SH97]). We denote with $p_{i,j}$ the pixel in position (i, j). The procedure for finding the largest rectangle with left upper corner (i, j) that matches a rectangle with left upper corner (k, h) is described in Fig. 1.

At the first step, the procedure computes the longest possible width for a rectangle match in (i, j) with respect to the position (k, h). The rectangle 1 x ℓ computed at the first step is the current rectangle match and the sizes of its sides are stored in $side1$ and $side2$. In order to check whether there is a better match than the current one, the longest one-dimensional match on the next row and column j, not exceeding the current width, is computed with respect to the row next to the current copy and to column h. Its length is stored in the temporary variable *width* and the temporary variable *length* is increased by one. If the rectangle R whose sides have size *width* and *length* is greater than the current match, the current match is replaced by R. We iterate this operation on each row until the area of the current match is greater or equal to the area of the longest feasible *width*-wide rectangle, since no further improvement would be possible at that point. For example, in Fig. 2 we apply the procedure to find the largest rectangle match between position $(0, 0)$ and $(6, 6)$.

$w = k$;

$r = i$;

$width = m$;

$length = 0$;

$side1 = side2 = area = 0$;

repeat

Let $p_{r,j} \cdots p_{r,j+\ell-1}$ be the longest match in (w, h) with $\ell \leq width$;

$length = length + 1$;

$width = \ell$;

$r = r + 1$;

$w = w + 1$;

if $(length * width > area)$ {

$area = length * width$;

$side1 = length$;

$side2 = width$;

}

until $area \geq width * (i - k + 1)$ or $w = i + 1$

Figure 1: Computing the largest rectangle match in (i, j) and (k, h).

A one-dimensional match of width 6 is found at step 1. Then, at step 2 a better match is obtained which is 2 x 4. At step 3 and step 4 the current match is still 2 x 4 since the longest match on row 3 and 9 has width 2. At step 5, another match of width 2 provides a better rectangle match which is 5 x 2. At step 6, the procedure stops since the longest match has width 1 and the rectangle match can cover at most 7 rows. It follows that 5 x 2 is the greedy match since a rectangle of width 1 cannot have a larger area. Obviously, this procedure can be used for computing the largest monochromatic rectangle in a given position (i, j) as well. If the 4 x 4 subarray in position (i, j) is monochromatic, then we compute the largest monochromatic rectangle in that position. Otherwise, we compute the largest rectangle match in the position provided by the hash table and update the table with the current position. If the subarray is not hashed to a pointer, then it is left uncompressed and added to the hash table with its current position. The positions covered by matches are skipped in the linear scan of the image.

As pointed out in [SH97], for typical bi-level images this scheme is extremely fast for square matches and there is no significant slowdown over simply reading and writing the image. As mentioned in the introduction, in [SH97] it is shown that the rectangle greedy matching technique requires $O(M \log M)$ time where M is the size of the match. Therefore, $O(n \log M)$ is the best sequential time for an image of size n if we compress by rectangle matching with M the maximum size of a rectangle. The encoding scheme is to precede each item with a flag field indicating whether there is a monochromatic square (0 for white, 10 for black), a match (110) or raw data (111). Pointers are encoded with the straightforward encoding with three integers for x, y and size while a simple variable-length code is used to specify the size of a monochromatic square. We also mentioned in the introduction that a key issue for

0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 1
0	0	1	1	1	0	0	1	0	0	0	0	0	1	0	step 2
1	0	1	1	1	0	0	1	0	1	0	0	1	1	1	step 3
0	1	1	0	1	1	0	0	0	0	0	0	0	1	1	step 4
0	1	1	0	1	0	0	1	0	0	0	0	0	0	1	step 5
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	step 6
0	0	1	0	1	1	0	0	1	0	1	1	1	1	1	step 1
0	0	1	0	1	1	0	0	1	1	0	0	1	1	1	step 2
0	0	1	0	1	1	1	0	0	0	0	0	1	1	1	step 3
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 4
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 5
0	0	1	0	1	1	0	1	0	0	0	0	1	1	1	step 6
0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	

Figure 2: The largest match in (0,0) and (6,6) is computed at step 5.

practical implementations is how pointers are encoded. As pointed out in [SH97], good pointer coding schemes are important for text compression and become even more important for images since the number of matches that are used is typically less than the number found and the straightforward coding uses many more bits per pointer. With rectangular matches this issue becomes even more significant. The encoding of monochromatic rectangles is a dominant factor of the compression performance and the efficiency of the method increases with large images.

In [CDG01] we experimented our rectangle greedy matching algorithm with a bounded size dictionary defined by a *window* comprising the last 64K pixels read. We bounded by twelve the number of bits to encode either the width or the length of a rectangle match. We use either four or eight or twelve bits to encode one rectangle side. Therefore, nine different kinds of rectangle are defined. A pointer is encoded in the following way:

- the flag field indicating the type of item;
- if the item is not monochromatic, sixteen bits which are raw or indicating the position of the match in the window;
- three or four bits encoding one of the nine kinds of rectangle;
- bits for the length and the width.

Larger rectangles are less frequent but still relevant for the compression performance. Then, four bits are used to indicate when twelve bits or eight and twelve bits are needed for the length and the width. This way of encoding rectangles plays a relevant role for the compression performance. In fact, it wastes four bits when twelve bits are required for the sides but saves four to twelve bits when four or eight bits suffice. On the CCITT bi-level image test set, we achieved 75 to 90 percent of the compression of JBIG1.

3 The Parallel Algorithm

To achieve logarithmic time we partition an m x n image I in x x y rectangular areas where x and y are $\Theta(\log^{1/2} mn)$. In parallel for each area, one processor applies the sequential parsing algorithm so that in logarithmic time each area will be parsed in rectangles, some of which are monochromatic. We do not allow overlapping of the monochromatic rectangles when we apply the sequential algorithm to each area. Each processor could work with a sliding window of size 64K and bounded matches, using the same pointer encoding scheme described in the previous section. However, before encoding we wish to compute larger monochromatic rectangles. If we compute unbounded monochromatic rectangles, the coding for them could be the flag field, log m bits for the length and log n bits for the width.

In the description of the algorithm, we use four m x n matrices RC, CC, W and L which are determined by the parsing procedure on each area. $RC[i,j]$ and $CC[i,j]$ are equal to zero if $I[i,j]$ is not covered by a monochromatic rectangle, otherwise they are equal to the row and column coordinate of the left upper corner of the monochromatic rectangle. $W[i,j]$ and $L[i,j]$ are equal to zero if $I[i,j]$ is not covered by a monochromatic rectangle, otherwise they are equal to the width and length of the monochromatic rectangle. We also use four matrices TRC, TCC, TW and TR to store temporary values needed for the computation of the final parsing, which are initially set to RC, CC, W and L. The procedure to compute larger monochromatic rectangles works as in Fig. 3.

Basically, we try to merge monochromatic rectangles adjacent on the horizontal boundaries and then on the vertical boundaries, doubling in this way the length and width of each area at each step. It is always the rectangle of an area in odd position with respect either to the vertical or horizontal order which tries to merge with the adjacent rectangle in the next area. Generally, this merging operation causes that the rectangles split into two or three subrectangles. The rectangle from which we start the merging is split in at most two subrectangles since we want to preserve the upper left corner. The merging is realized by updating the temporary matrices storing the information on the monochromatic rectangles computed on the image. If we obtain a larger rectangle then we update the original matrices, otherwise we continue merging by working with the temporary values to see if we can get a larger rectangle later.

We describe the procedure more in details. At the first line internal to the main loop (Figure 3), we consider in parallel the left lower corners of monochromatic rectangles of the areas in odd positions which are adjacent to a monochromatic rectangle with the same color in the next area below. Then, at line 3 we change the width and length of the rectangle considered, where the length is the sum of the lengths of the two adjacent rectangles and the width changes if the right corners of the rectangle in the next area are to the left of the right corners of the other rectangle. At line 4 and 5 the values in the temporary matrices are changed also for the pixels in the next area since they merged. Obviously, these changes can be made with optimal work in logarithmic time. As mentioned above, the merging causes a splitting of rectangles into subrectangles and the values in the temporary matrices must be redefined also

until $x > m/2$ or $y > n/2$ do {
1 **in parallel for** i and j: $i = ax$, a odd, $j = TCC[i,j]$, $TRC[i+1,j] <> 0$ and $I[i,j] = I[i+1,j]$ {
2 **in parallel for** $TRC[i,j] \leq k \leq i$, $j \leq h \leq \min\{TCC[i+1,j] + TW[i+1,j] - 1, j + TW[i,j] - 1\}$ {
3 $TW[k,h] = \min\ \{TCC[i+1,j] + TW[i+1,j] - 1, j + TW[i,j] - 1\} - j + 1;$
3 $TL[k,h] = TL[k,h] + TL[i+1,j];$ }
4 **in parallel for** $i{+}1 \leq k \leq i{+}TL[i{+}1,j]$, $j \leq h \leq \min\{TCC[i{+}1,j]{+}TW[i{+}1,j]{-}1, j{+}TW[i,j]{-}1\}$
5 { $TCC[k,h] = j; TRC[k,h] = TRC[i,j]; TW[k,h] = TW[i,j]; TL[k,h] = TL[i,j];$ } }
6 **in parallel for** i and j: $TCC[i,j-1] + TW[i,j-1] = j, TCC[i,j] <> j, TCC[i,j] <>= 0$ or
7 $TCC[i,j] = j$ {
8 compute the smallest k with $k \geq j$: $TCC(i,k) + TW[i,k] > k+1, TCC[i,k+1] = k+1$ or
9 $TCC(i,k) + TW[i,k] = k+1$
10 **in parallel for** $j \leq h \leq k$ { $TCC[i,h] = j$; $TW[i,h] = k - j + 1$; } }
11 **in parallel for** i, j: $TRC[i,j] = i$ and $TCC[i,j] = j$
12 **if** $TW[i,j] * TL[i,j] \geq W[i,j] * L[i,j]$ **in parallel for** $i \leq k < i + TL[i,j]$, $j \leq h < j + TW[i,j]$ {
13 $RC[i,j] = TRC[i,j]$; $CC[i,j] = TCC[i,j]$; $W[i,j] = TW[i,j]$; $L[i,j] = TL[i,j]$; }
14 **in parallel for** i and j: $j = ay$, a odd, $i = TRC[i,j]$, $TRC[i,j+1] <> 0$ and $I[i,j] = I[i,j+1]$ {
15 **in parallel for** $TCC[i,j] \leq h \leq j$, $i \leq k \leq \min\{TRC[i,j+1] + TL[i,j+1] - 1, i + TL[i,j] - 1)\}$ {
16 $TL[k,h] = \min\{TRC[i,j+1] + TL[i,j+1] - 1, i + TL[i,j] - 1))\};$
16 $TL[k,h] = TL[k,h] + TL[i+1,j];$ }
17 **in parallel for** $j{+}1 \leq h \leq j{+}TW[i,j{+}1]$, $i \leq k \leq \min\{TRC[i,j{+}1]{+}TL[i,j{+}1]{-}1, i{+}TL[i,j]{-}1)\}$
18 { $TRC[k,h] = i; TLC[k,h] = TLC[i,j]; TW[k,h] = TW[i,j]; TL[k,h] = TL = [i,j];$ } }
19 **in parallel for** i and j: $TRC[i-1,j] + TL[i-1,j] = i-1, TRC[i,j] <> i, TRC[i,j] <>= 0$ or
20 $TRC[i,j] = i$ {
21 compute the smallest k with $k \geq i$: $TRC(k,j) + TL[k,j] > k+1, TRC[k+1,j] = k+1$ or
22 $TRC(k,j) + TL[k,j] = k+1$
23 **in parallel for** $i \leq h \leq k$ { $TRC[h,j] = i$; $TW[h,j] = k - i + 1$; } }
24 **in parallel for** i, j: $i = TRC[i,j]$ and $j = TRC[i,j]$
25 **if** $TW[i,j] * TL[i,j] \geq W[i,j] * L[i,j]$ **in parallel for** $i \leq k < i + TL[i,j]$, $j \leq h < j + TW[i,j]$ {
26 $RC[i,j] = TRC[i,j]$; $CC[i,j] = TCC[i,j]$; $W[i,j] = TW[i,j]$; $L[i,j] = TL[i,j]$; }
27 $x = 2x$; $y = 2y$; }

Figure 3: How to compute monochromatic rectangles in parallel.

for the pixels covered by the other rectangles produced by the merging. This is done from line 6 to 10. In parallel we consider all the pixels for which, according to the temporary values, either they are not on the leftmost column of a rectangle and the adjacent pixels in front of them result to be on the rightest column of a rectangle (line 6) or they are on the leftmost column (line 7). For each of them, we compute the closest pixel to the right for which, according to the temporary values, either it is not on the rightmost column of a rectangle and the next pixel results to be on the leftmost column of a rectangle (line 8) or it is on the rightmost column of a rectangle (line 9). Being this pixel the closest to the one computed in lines 6 and 7, they must be on the rightmost and leftmost column of the same monochromatic rectangle respectively. This is redefined in the temporary matrices at line 10. At this point, for each left upper corner of a monochromatic rectangle (line 11) if we obtained a larger rectangle after the merging (line 12) we can ovewrite the information on the new rectangle on the original matrices (line 13). Observe that this way of updating the matrices may introduce overlapping of the monochromatic rectangles. Then, we repeat the same procedure trying to merge rectangles horizontally (line 14–26).

To analize the complexity of the algorithm, it is enough to consider that at each iteration of the main loop we double the sides of the areas and to recall the classical parallel prefix computation technique. All the statements inside the loop require logarithmic time with optimal parallel work (lines 8–9 and 21–22 by parallel prefix). Since no operation is executed if there is nothing to merge, the total running time with optimal parallel work is $O(\log n \log M)$, where M is the maximum size of a monochromatic rectangle. Then, from the matrices we can easily derive the sequence of pointers with optimal parallel work and logarithmic time by parallel prefix.

4 Conclusions

In this paper, we showed a work-optimal parallel algorithm for lossless image compression by block matching using a rectangle greedy matching technique which requires $O(\log M \log n)$ time. The algorithm is suitable for an implementation on practical parallel architectures as meshes of trees, multigrids and pyramids.

As future work, a detailed study on how the algorithm must be implemented on these architectures could be provided. Also, practical parallel algorithms for decompression should be designed.

References

[BCW90] Bell T.C., Cleary J.G., Witten I.H: Text Compression. Prentice Hall.

[CDG01] Cinque L., De Agostino S., Grande E: LZ1 Compression of Binary Images using a Simple Rectangle Greedy Matching Technique. IEEE Data Compression Conference, 492.

[CDL02] Cinque L., De Agostino S., Liberati F.: A Parallel Algorithm for Lossless Image Compression by Block Matching. IEEE Data Compression Conference, 450.

[LZ76] Lempel A., Ziv J: On the Complexity of Finite Sequences. IEEE Transactions on Information Theory, 22, 75-81.

[LZ77] Lempel A., Ziv J: A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, 23, 337-343.

[St88] Storer J.A.: Data Compression: Methods and Theory. Computer Science Press.

[S96] Storer J. A.: Lossless Image Compression using Generalized LZ1-Type Methods. IEEE Data Compression Conference, 290-299.

[SH97] Storer J. A., Helfgott H.: Lossless Image Compression by Block Matching. The Computer Journal, 40, 137-145.

[SS82] Storer J. A., Szymanski T. G.: Data Compression via Textual Substitution. Journal of ACM, 29, 928-951.

[ZL78] Ziv J., Lempel A.,: Compression of Individual Sequences via Variable Rate Coding. Transactions on Information Theory, 24, 530-536.

A Note on Randomized Algorithm for String Matching with Mismatches

Kensuke Baba, Ayumi Shinohara,
Masayuki Takeda, Shunsuke Inenaga, and Setsuo Arikawa

Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan

e-mail: {baba, ayumi, takeda, s-ine, arikawa}@i.kyushu-u.ac.jp

Abstract. Atallah et al. [ACD01] introduced a randomized algorithm for string matching with mismatches, which utilized fast Fourier transformation (FFT) to compute convolution. It estimates the score vector of matches between text string and a pattern string, i.e. the vector obtained when the pattern is slid along the text, and the number of matches is counted for each position. In this paper, we simplify the algorithm and give an exact analysis of the variance of the estimator.

Key words: Pattern matching, mismatch, FFT, convolution, randomized algorithm

1 Introduction

Let $T = t_1, \ldots, t_n$ be a text string and $P = p_1, \ldots, p_m$ be a pattern string over an alphabet Σ. *String matching problem* is to find all occurrences of the pattern P in the text T. *Approximate string matching problem* is to find all occurrences of small variations of the original pattern P in the text T. Substitution, insertion, and deletion operations are often allowed to introduce the variations. In this paper, we allow the substitution operation only. The derived problem is usually called *string matching with mismatches*. It is essentially to compute the *score vector* $C(T, P) = (c_1, \ldots, c_{n-m+1})$ between T and P, where each c_i counts the number of matches between the substring $t_i, \ldots, t_{i+m-1}$ of the text T and the pattern P. If $c_i = m$, the pattern exactly occurs at position i in the text. Fig. 1 shows an example of the score vector. A reasonable amount of effort has been paid for this problem [Abr87, BYG92, BYP96, FP74, Kar93]. Refer the textbooks [CR94, Gus97] to know the history and various results.

Recently, Atallah et al. [ACD01] introduced a randomized algorithm of Monte-Carlo type which returns an estimation of the score vector $C(T, P)$. The estimation is performed by averaging independent equally distributed estimates. Let k be the number of ramdomly sampled estimations, then the time complexity is $O(kn \log m)$ by utilizing a fast Fourier transformation (FFT). They showed that the expected value of the estimation is equal to the score vector, and that the variance is bounded by $(m - c_i)^2/k$.

In this paper, we give a slight simplification of their algorithm. Moreover, we analyze the variance of the estimator exactly.

i	1	2	3	4	5	6	7	8	9	10
text	a	c	b	a	b	b	a	c	c	b
pattern	$\underline{\mathtt{a}}$	b	$\underline{\mathtt{b}}$	$\underline{\mathtt{a}}$	c					
		a	$\underline{\mathtt{b}}$	b	a	c				
			a	b	$\underline{\mathtt{b}}$	a	c			
				$\underline{\mathtt{a}}$	$\underline{\mathtt{b}}$	$\underline{\mathtt{b}}$	$\underline{\mathtt{a}}$	$\underline{\mathtt{c}}$		
					a	$\underline{\mathtt{b}}$	b	a	$\underline{\mathtt{c}}$	
						a	b	b	a	c
c_i	3	1	1	5	2	0				

Figure 1: Score vector between the text `acbabbaccb` and the pattern `abbac`.

2 Preliminaries

Let $\mathcal{N}$ be the set of non-negative integers. Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. We denote the cardinality of a set S by $|S|$ or $\#S$.

We define a function δ from $\Sigma \times \Sigma$ to $\{0, 1\}$ by

$$\delta(a, b) = \begin{cases} 1 & \text{if } a = b, \\ 0 & \text{if } a \neq b. \end{cases}$$

For a text string $T = t_1 t_2 \dots t_n$ and a pattern string $P = p_1 p_2 \dots p_m$, *the score vector of matches between T and P* is defined as $C(T, P) = (c_1, c_2, \dots, c_{n-m+1})$, where $c_i = \sum_{j=1}^{m} \delta(t_{i+j-1}, p_j)$. That is, c_i is the number of matches between the text and the pattern when the first letter of the pattern in positioned in front of the ith letter of the string.

3 Deterministic Algorithm

In this section, we introduce a deterministic algorithm to compute the score vector for given text T and pattern P. Although it might not be practical for large alphabet, it will be a base for the randomized algorithm explored in the next section.

3.1 Binary Alphabet Case

We first consider a binary alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. We define a function $\psi : \Sigma \to \{-1, 1\}$ by $\psi(\mathtt{a}) = 1$ and $\psi(\mathtt{b}) = -1$. By using ψ, we convert the strings T and P into the sequences of integers as follows.

$$\begin{aligned} \psi(T) &= \psi(t_1), \psi(t_2), \dots\dots\dots, \psi(t_n), \\ \psi(P) &= \psi(p_1), \psi(p_2), \dots, \psi(p_m). \end{aligned}$$

Let $A^{\psi}(T, P) = (a_1^{\psi}, a_2^{\psi}, \dots, a_{n-m+1}^{\psi})$ where $a_i^{\psi} = \sum_{j=1}^{m} \psi(t_{i+j-1}) \cdot \psi(p_j)$.

Lemma 1 *For any* $1 \leq i \leq n-m+1$, $c_i = (a_i^\psi + m)/2$.

Proof. Since $c_i = \#\{j \mid t_{i+j-1} = p_j,\ 1 \leq j \leq m\}$, we have $a_i^\psi = \#\{j \mid t_{i+j-1} = p_j,\ 1 \leq j \leq m\} - \#\{j \mid t_{i+j-1} \neq p_j,\ 1 \leq j \leq m\} = c_i - (m - c_i) = 2c_i - m$. Thus $c_i = (a_i^\psi + m)/2$. □

The above lemma implies that we have only to compute $A^\psi(T, P)$ to get the score vector $C(T, P)$. Since the sequence $A^\psi(T, P)$ is the convolution of $\psi(T)$ with the reverse of $\psi(P)$, we can calculate all the a_i's simultaneously by the use of fast Fourier transform (FFT) in $O(n \log m)$ time as follows. As is stated in [ACD01], we additionally apply the standard technique [CR94] of partitioning the text into overlapping chunks of size $(1+\alpha)m$ each, and then processing each chunk separately. Processing one chunk gives us αm components of C. Since we have $n/(\alpha m)$ chunks and each chunk can be computed in $O((1+\alpha)m \log((1+\alpha)m))$ by FFT, the total time complexity is $\frac{n}{\alpha m} \cdot O((1+\alpha)m \log((1+\alpha)m)) = O\left(\frac{(1+\alpha)}{\alpha} n \log((1+\alpha)m)\right) = O(n \log m)$ by choosing $\alpha = O(m)$.

Theorem 1 *For a binary alphabet, the score vector* C *can be exactly computed in* $O(n \log m)$ *time.*

3.2 General Case

We now consider general case $|\Sigma| > 2$. Let Ψ_Σ be the set of all mappings from Σ to $\{-1, 1\}$. Remark that $|\Psi_\Sigma| = 2^{|\Sigma|}$. We abbreviate Ψ_Σ with Ψ when Σ is clear from the context. The next lemma is obvious.

Lemma 2 *For any* $\psi \in \Psi_\Sigma$ *and any* $a, b \in \Sigma$,

$$\psi(a) \cdot \psi(b) = \begin{cases} 1 & \text{if } \psi(a) = \psi(b), \\ -1 & \text{if } \psi(a) \neq \psi(b). \end{cases}$$

Lemma 3 *For any* $a, b \in \Sigma$,

$$\frac{1}{|\Psi|} \sum_{\psi \in \Psi} \psi(a) \cdot \psi(b) = \delta(a, b).$$

Proof. In case of $a = b$, then $\psi(a) = \psi(b)$ for any $\psi \in \Psi$. Therefore $\psi(a) \cdot \psi(b) = 1$ for any ψ by Lemma 2, and the sum $\sum_{\psi \in \Psi} \psi(a) \cdot \psi(b)$ equals to the cardinality of Ψ. Thus, the left side of the equation is unity.

To prove the lemma in case of $a \neq b$, we show a more general proposition:

$$\sum_{\psi \in \Psi} \psi(d_1) \cdot \cdots \cdot \psi(d_n) \cdot \psi(b) = 0 \quad \text{if } d_1 \neq b, \cdots, d_n \neq b \ (n \geq 0).$$

By the assumption that b is distinct from $d_1, \cdots, d_n$,

$$\begin{aligned} & \sum_{\psi \in \Psi} \psi(d_1) \cdot \cdots \cdot \psi(d_n) \cdot \psi(b) \\ = & \sum_{\psi(b)=1, \psi \in \Psi} \psi(d_1) \cdot \cdots \cdot \psi(d_n) \cdot 1 + \sum_{\psi(b)=-1, \psi \in \Psi} \psi(d_1) \cdot \cdots \cdot \psi(d_n) \cdot (-1) \\ = & \ 0. \end{aligned}$$

Thus, by the proposition for $n = 1$, the left side of the equation is zero. □

Theorem 2 *For any $1 \le i \le m - n + 1$,*

$$c_i = \frac{1}{|\Psi|}\sum_{\psi\in\Psi} a_i^{\psi}. \tag{1}$$

Proof. By the definition of a_i^{ψ} and Lemma 3, the right side of the equation can be changed as follows.

$$\begin{aligned}\frac{1}{|\Psi|}\sum_{\psi\in\Psi} a_i^{\psi} &= \frac{1}{|\Psi|}\sum_{\psi\in\Psi}\sum_{j=1}^{m}\psi(t_{i+j-1})\cdot\psi(p_j)\\ &= \sum_{j=1}^{m}\frac{1}{|\Psi|}\sum_{\psi\in\Psi}\psi(t_{i+j-1})\cdot\psi(p_j)\\ &= \sum_{j=1}^{m}\delta(t_{i+j-1},p_j).\end{aligned}$$

Since the last formula is the definition of c_i, the theorem is proved. □

Theorem 3 *$C(T, P)$ can be exactly computed in $O(2^{|\Sigma|} n \log m)$ time.*

Proof. By Theorem 2 c_i is the mean of a_i^{ψ} for every $\psi \in \Psi_\Sigma$, therefore $C(T, P)$ is obtained by computing all $A^{\psi}(T, P)$. Since each $A^{\psi}(T, P)$ can be computed in $O(n \log m)$ time, we can calculate $C(T, P)$ in $O(2^{|\Sigma|} n \log m)$ time. □

We note that if the alphabet Σ is infinite, by splitting the text in chunks of length $O(m)$ to be dealt with independently ensures it will work with an alphabet size $O(m)$, so that $C(T, P)$ can be exactly computed in $O(2^{O(m)} n \log m)$.

4 Randomized Algorithm

A shortcoming of the deterministic algorithm in the last section is that the running time is exponential with respect to the size of alphabet. It is not practical for large alphabet. In this section, we propose a randomized algorithm which was inspired by Atallah et al. [ACD01].

Let us noticed that Theorem 2 can be interpreted as follows. Each c_i is the mean of random variable $X_i = \sum_{j=1}^{m} \psi(t_{i+j-1}) \cdot \psi(p_j)$, assuming that ψ is drawn uniformly randomly from Ψ. The observation leads us to the following randomized algorithm. Instead of computing all vectors $A_\psi(T, P) = (a_1^{\psi}, a_2^{\psi}, \ldots, a_{n-m+1}^{\psi})$ where $a_i^{\psi} = \sum_{j=1}^{m} \psi(t_{i+j-1}) \cdot \psi(p_j)$ to average them, we compute only k samples of them for randomly chosen $\psi_1, \ldots, \psi_k \in \Psi$. Since the expected value of X_i equals to c_i, it will give a good estimation for large enough k. We will give a formal proof of it, and exactly analyze the variance of X_i in the sequel. Fig. 2 illustrates the core part of the algorithm for the basic case $n = (1 + \alpha)m$.

We now analyze the mean and the variance of the estimator $\hat{c}_i$. Since all the random variable $\hat{c}_i$ are defined in a similar way, we generically consider the random variable

$$\hat{s} = \frac{1}{k}\sum_{\ell=1}^{k}\sum_{j=1}^{m}\psi(t_j)\cdot\psi(p_j)$$

Procedure ESTIMATESCORE
Input: a text $T = t_1 \dots t_{(1+\alpha)m}$ and a pattern $P = p_1 \dots p_m$ in Σ^*.
Output: an estimate for the score vector $C(T, P)$.
for $\ell := 1$ **to** k **do begin**
randomly and uniformly select a ψ_ℓ from Ψ_Σ.
Let $T_\ell = \psi_\ell(T)$. Note that T_ℓ is a sequence over $\{-1, 1\}$ of length $(1+\alpha)m$.
Let P_ℓ be the concatenation of $\psi_\ell(P)$ with trailing αm zeros.
compute the vector C_ℓ as the convolution of T_ℓ with the reverse of P_ℓ by FFT.
end
compute the vector $\hat{C} = \frac{1}{k}\sum_{\ell=1}^{k} C_\ell$ and output it as an estimate of $C(T, P)$.

Figure 2: Randomized Algorithm

where the t_j's and the p_j's are fixed and mapping ψ's are independently and uniformly selected from Ψ_Σ. The definition implies that $\hat{s}$ is the mean of k random variables which are drawn from independent and identical distribution. The random variable can be defined by

$$s = \sum_{j=1}^{m} \psi(t_j) \cdot \psi(p_j),$$

and the mean $E(\hat{s})$ and variance $V(\hat{s})$ are

$$E(\hat{s}) = E(s) \quad \text{and} \quad V(\hat{s}) = \frac{V(s)}{k}.$$

The number c of matches between $T = t_1 \dots t_m$ and $P = p_1 \dots p_m$ is

$$c = \sum_{j=1}^{m} \delta(t_j, p_j).$$

Lemma 4 *The mean of $\hat{s}$ is equal to c.*

Proof. By Lemma 3,

$$\begin{aligned} E(\hat{s}) = E(s) &= \frac{1}{|\Psi|}\sum_{\psi\in\Psi} s \\ &= \frac{1}{|\Psi|}\sum_{\psi\in\Psi}\sum_{j=1}^{m} \psi(t_j)\cdot\psi(p_j) \\ &= \sum_{j=1}^{m}\frac{1}{|\Psi|}\sum_{\psi\in\Psi} \psi(t_j)\cdot\psi(p_j) \\ &= \sum_{j=1}^{m} \delta(t_j, p_j). \end{aligned}$$

Thus, the mean of $\hat{s}$ is c. $\square$

In order to analyze the variance of s accurately, we introduce the following function $\rho_{T,P} : \Sigma \times \Sigma \to \mathcal{N}$ depending on text $T = t_1 \ldots t_m$ and pattern $P = p_1 \ldots p_m$, which give a statistics of T and P.

$$\rho_{T,P}(a,b) = \#\{j \mid t_j = a \text{ and } p_j = b,\ 1 \le j \le m\}$$

For example, let $T = \texttt{aabac}$ and $P = \texttt{abbba}$. Then $\rho_{T,P}(\texttt{a},\texttt{b}) = 2$, $\rho_{T,P}(\texttt{a},\texttt{a}) = \rho_{T,P}(\texttt{b},\texttt{b}) = \rho_{T,P}(\texttt{c},\texttt{a}) = 1$, and the others are zero. We omit the subscription T, P of $\rho_{T,P}$ in the sequel. In addition, we use the following expression.

$$\tau(a,b) = \rho(a,b) + \rho(b,a).$$

The next lemma is obvious from the definition.

Lemma 5 $\displaystyle\sum_{(a,b)\in\Sigma\times\Sigma} \rho(a,b) = \frac{1}{2} \sum_{(a,b)\in\Sigma\times\Sigma} \tau(a,b) = m.$

The next lemma gives the exact variance of $\hat{s}$, in terms of ρ.

Lemma 6 *The variance of $\hat{s}$ is*

$$V(\hat{s}) = \frac{1}{k} \sum_{a \ne b} \left(\rho(a,b)^2 + \rho(a,b) \cdot \rho(b,a) \right).$$

Proof. Since the mean of s equals to c by Lemma 4,

$$V(\hat{s}) = \frac{1}{k} V(s) = \frac{1}{k} \frac{1}{|\Psi|} \sum_{\psi \in \Psi} (s-c)^2.$$

By the definition of ρ,

$$\begin{aligned}
s &= \sum_{(a,b)\in\Sigma\times\Sigma} \psi(a)\cdot\psi(b)\cdot\rho(a,b) \\
&= \sum_{a=b} \rho(a,b) + \sum_{a\ne b} \psi(a)\cdot\psi(b)\cdot\rho(a,b), \text{ and} \\
c &= \sum_{a=b} \rho(a,b).
\end{aligned}$$

Therefore,

$$\begin{aligned}
\frac{1}{|\Psi|} \sum_{\psi\in\Psi} (s-c)^2 &= \frac{1}{|\Psi|} \sum_{\psi\in\Psi} \left(\left(\sum_{a=b} \rho(a,b) + \sum_{a\ne b} \psi(a)\cdot\psi(b)\cdot\rho(a,b) \right) - \sum_{a=b} \rho(a,b) \right)^2 \\
&= \frac{1}{|\Psi|} \sum_{\psi\in\Psi} \left(\sum_{a\ne b} \psi(a)\cdot\psi(b)\cdot\rho(a,b) \right)^2 \\
&= \frac{1}{|\Psi|} \sum_{\psi\in\Psi} \left(\sum_{a\ne b} \psi(a)\cdot\psi(b)\cdot\rho(a,b) \right) \left(\sum_{a'\ne b'} \psi(a')\cdot\psi(b')\cdot\rho(a',b') \right) \\
&= \frac{1}{|\Psi|} \sum_{\psi\in\Psi} \sum_{a\ne b} \sum_{a'\ne b'} \psi(a)\cdot\psi(b)\cdot\rho(a,b)\cdot\psi(a')\cdot\psi(b')\cdot\rho(a',b') \\
&= \sum_{a\ne b} \left(\rho(a,b) \cdot \sum_{a'\ne b'} \rho(a',b') \frac{1}{|\Psi|} \sum_{\psi\in\Psi} \psi(a)\cdot\psi(b)\cdot\psi(a')\cdot\psi(b') \right).
\end{aligned}$$

Let us take $\alpha(a,b,a',b') = \dfrac{1}{|\Psi|}\sum_{\psi\in\Psi}\psi(a)\cdot\psi(b)\cdot\psi(a')\cdot\psi(b')$, and show that

$$\alpha(a,b,a',b') = \begin{cases} 1 & \text{if either } a=a' \text{ and } b=b', \text{ or } a=b' \text{ and } a'=b, \\ 0 & \text{otherwise,} \end{cases}$$

by the case analysis whether there exists a distinct character from the others in a,b,a',b'. If there exists such a character, then $\alpha(a,b,a',b') = 0$ by the proof of Lemma 3. If there does not exist such a character, then we have either $a=a'$ and $b=b'$, or $a=b'$ and $b=a'$ by the assumption that both $a\neq b$ and $a'\neq b'$. Then, by Lemma 3 and the fact that $\psi(a)^2 = 1$ for any $\psi\in\Psi$ and any $a\in\Sigma$ since $\psi(a)\in\{-1,1\}$,

$$\alpha(a,b,a',b') = \frac{1}{|\Psi|}\sum_{\psi\in\Psi}\psi(a)^2\cdot\psi(b)^2 = 1.$$

Thus,

$$\begin{aligned} V(\hat{s}) &= \frac{1}{k}\sum_{a\neq b}\rho(a,b)\left(\rho(a,b)+\rho(b,a)\right) \\ &= \frac{1}{k}\sum_{a\neq b}\left(\rho(a,b)^2+\rho(a,b)\cdot\rho(b,a)\right). \end{aligned}$$

□

Moreover, by the definition of τ, we have

$$\begin{aligned} \sum_{a\neq b}\left(\rho(a,b)^2+\rho(a,b)\cdot\rho(b,a)\right) &= \frac{1}{2}\sum_{a\neq b}\left(\rho(a,b)^2+2\rho(a,b)\cdot\rho(b,a)+\rho(b,a)^2\right) \\ &= \frac{1}{2}\sum_{a\neq b}\left(\rho(a,b)+\rho(b,a)\right)^2 \\ &= \frac{1}{2}\sum_{a\neq b}\tau(a,b)^2 \\ &= \sum_{a<b}\tau(a,b)^2. \end{aligned}$$

Therefore, the variance can be exactly restated in term of τ as follows, which might be more intuitive.

Theorem 4 *The variance of $\hat{s}$ is*

$$V(\hat{s}) = \frac{1}{k}\sum_{a<b}\tau(a,b)^2.$$

Remind that $\tau(a,b)$ represented the number of positions $j = 1,\ldots,m$ in T and P, such that (t_j,p_j) is either (a,b) or (b,a). If T exactly matches P, then $V(\hat{s}) = 0$, which implies that the estimation is always m, without any error. On the other hand, since $\sum_{a<b}\tau(a,b) = m-c$, the variance $V(\hat{s})$ is maximized for inputs which have no match and are constructed by only two characters, for example, $T =$ `aaaaaa`, $P =$ `bbbbbb`, and $T =$ `aaabba`, $P =$ `bbbaab`.

We now state the bound of the variance of $\hat{s}$ in terms of m and c, that exactly fits to the one proved by Atallah et al. [ACD01].

Lemma 7 *The variance of $\hat{s}$ is bounded as follows.*

$$V(\hat{s}) \leq \frac{(m-c)^2}{k}.$$

Proof. By Lemma 5,

$$\begin{aligned} m - c &= \sum_{(a,b)\in\Sigma\times\Sigma} \rho(a,b) - \sum_{a=b} \rho(a,b) \\ &= \sum_{a\neq b} \rho(a,b) \\ &= \frac{1}{2}\sum_{a\neq b} \tau(a,b) \\ &= \sum_{a<b} \tau(a,b). \end{aligned}$$

Therefore, by Theorem 4,

$$\begin{aligned} \frac{(m-c)^2}{k} - V(\hat{s}) &= \frac{1}{k}\left(\sum_{a<b} \tau(a,b)\right)^2 - \frac{1}{k}\sum_{a<b} \tau(a,b)^2 \\ &= \frac{1}{k}\sum_{a<b}\left(\tau(a,b)\cdot {\sum_{a'<b'}}^{*} \tau(a',b')\right), \end{aligned}$$

where ${\sum_{a'<b'}}^{*} \tau(a',b')$ expresses the sum of $\tau(a',b')$ except for the two cases $a' = a, b' = b$ and $a' = b, b' = a$. Since $\tau(a,b) \geq 0$ for any a and b, the last formula is not less than zero. □

We now have the main theorem.

Theorem 5 *Algorithm* EstimateScore *runs in $O(kn\log m)$ time. The mean of the estimation equals to the score vector C, and the variance of each entry is bounded by $(m-c_i)^2/k$.*

5 Conclusion

We gave a randomized algorithm for string matching with mismatches, which can be regarded as a slight simplification of the one due to Atallah et al. [ACD01]. For comparison, we give a brief description of their algorithm. It treats the set Ψ' of all mappings from Σ to $\{0, 1, \ldots, |\Sigma| - 1\}$, and the basic equation is

$$c_i = \frac{1}{|\Psi'|}\sum_{\psi\in\Psi'}\sum_{j=1}^{m} \omega^{\psi(t_{i+j-1})-\psi(p_j)}, \tag{2}$$

where ω is a primitive $|\Sigma|$th root of unity. When $|\Sigma| = 2$, we know $\omega = -1$, and that the equation (2) directly corresponds to the equation (1) in ours. The difference is how to treat general alphabet $|\Sigma| > 2$. In our algorithm, the converted sequence $\psi(T)$

is simply over $\{-1, 1\}$, while in their algorithm $\psi(T)$ is over $\{1, \omega, \omega^2, \ldots, \omega^{|\Sigma|-1}\}$ that are complex numbers. When computing the convolution by FFT, the computation of the former will be much simpler (and possibly faster) than the latter. From the view point of the precision of the numerical calculations, the former might be preferable to the latter, although we have not yet studied explicitly. Moreover, this simplification enabled us to reach the exact estimation of the variance (Theorem 4), by fairly primitive discussion. An interesting point is that the variance is still independent from the size of alphabet, although we map Σ into $\{-1, 1\}$, instead of $\{0, 1, \ldots, |\Sigma| - 1\}$.

In their paper [ACD01], they considered various extensions, such as string matching with classes, class components, "never match" and "always match" symbols, weighted case, and higher dimension arrays. We think our simplification will be valid without any difficulty for all those extensions, although we have not completely verified them yet.

References

[Abr87] K. Abrahamson. Generalized string matching. *SIAM Journal on Computing*, 16(6):1039–1051, 1987.

[ACD01] M. J. Atallah, F. Chyzak, and P. Dumas. A randomized algorithm for approximate string matching. *Algorithmica*, 29:468–486, 2001.

[BYG92] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35:74–82, 1992.

[BYP96] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. *Information Processing Letters*, 59(1):21–27, 1996.

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

[FP74] M. J. Fischer and M. S. Paterson. String-matching and other prodeucts. In *Complexity of Computation (Proceedings of the SIAM-AMS Applied Mathematics Symposium, New York, 1973)*, pages 113–125, 1974.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.

[Kar93] H. Karloff. Fast algorithms for approximately couting mismatches. *Information Processing Letters*, 48(2):53–60, 1993.

A Recursive Function for Calculating the Number of Legal Strings of Parentheses and for Calculating Catalan Numbers

Kirke Bent

Parallel Business Software
29 Pine Street
Chatham, NJ 07928, USA

e-mail: `parbzsft@bellatlantic.net`

Abstract. This paper discusses the number of legal strings of n pairs of parentheses as well as a structure of the set of these strings. As the number of such strings is known to be the Catalan number, a structure of Catalan numbers is thereby developed. A recursive function is developed that counts the set and calculates the Catalan number. The function uses two parameters and is thus a generalization of Catalan numbers.

Key words: Parenthetical strings, recursive functions, stringology, combinatorics, generalized Catalan numbers.

1 Introduction

This paper concerns the problem of calculating the number of legal strings of parentheses that can be constructed from n pairs of parentheses. This number is known to be the Catalan number. There is a large literature of Catalan number interpretations and connections [2, 3, 4, 5, 6, 7]. Stanton and White have a proof of the correspondence between Catalan numbers and legal parenthetical strings[7]. The Catalan number is defined as

$$C_n = \binom{2n}{n} \div (n+1).$$

The ordinary meaning of "legal strings" of parentheses is intended here: 1) The strings are conventionally constructed from left to right. 2) At any point in the string, the number of left parentheses is equal to or greater than the number of right parentheses. 3) all of the $2n$ parentheses are used.

For example, $C_3 = 5$; the legal strings of 3 pairs of parentheses are

$$(\,(\,(\,)\,)\,),\ \ (\,(\,)\,(\,)\,),\ \ (\,(\,)\,)(\,),\ \ (\,)\,(\,(\,)\,),\ \text{and}\ (\,)\,(\,)\,(\,).$$

The paper offers a way to calculate Catalan numbers with a recursive function and a structure of the strings and the number.

2 Outline

Four areas emerge from consideration of this function:

2.1 A Chart

This is a chart of the construction of the C_n legal parenthetical strings composed of n pairs of parentheses. The number of such strings is an interpretation of Catalan numbers. The chart can be interpreted as a rooted tree. Evaluation of the function counts the leaves of the tree.

2.2 A Function

The function, denoted here by $B_{n,m}$, uses two parameters. The n^{th} Catalan number, C_n, is produced by $B_{n,0}$. The domain of both parameters of $B_{n,m}$ is the non-negative integers. In the recursive descent, m takes on values both higher and lower than n.

2.3 A Generalization

This generalization of Catalan numbers is based on the two parameters. It includes C_n.

2.4 A Structure

This structure of Catalan numbers is suggested by the chart but can be expressed algebraically.

3 Elaboration

3.1 The Chart

The idea behind the chart is simply writing the legal parenthetical expressions according to the definition above.

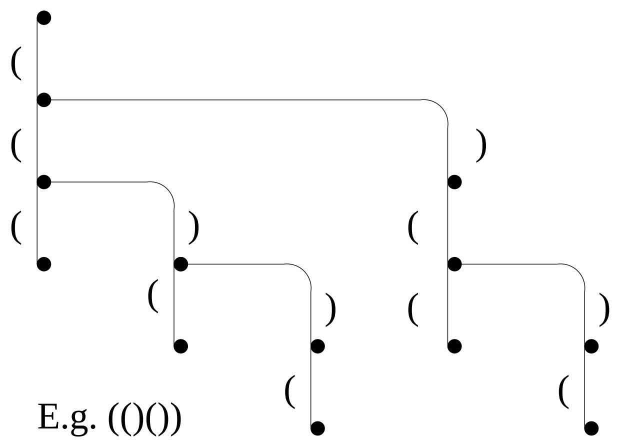


Figure 1: Forming all legal arrangements of 3 pairs of parentheses

Consider this as a rooted tree. Each edge represents adding a parenthesis. If there are two edges descending from a vertex, then there is a choice of adding a left or right parenthesis at that point. By following all paths from the root to a leaf, all legal expressions have been written. Note that final right parentheses are not needed to count leaves.

The steps in drawing the chart are:

1. Start at the top with n pairs of parentheses.

2. Stop if there are no more left parentheses.

3. Draw a vertical line downwards. This represents a left parenthesis and "uses" one. If the number of left parentheses used (before this one was drawn) exceeds the number of right ones used, draw another line from the same starting point but to the right and then curving downwards. This represents a right parenthesis and uses one.

4. Repeat steps 2, 3, and 4 for each end point.

The vertex at the botton of each line drawn represents the parenthetical string as constructed so far.

These conventions are somewhat arbitrary, as conventions must be, but they result in a picture that is regular and easy to understand. The chart was helpful in defining the function and discovering the structure.

3.2 The Function

$$B_{n,m} = \begin{cases} B_{n-1,m+1} + B_{n,m-1} & if \quad (n > 0) \wedge (m > 0) \\ B_{n-1,m+1} & if \quad (n > 0) \wedge (m = 0) \\ 1 & if \quad (n = 0) \end{cases}$$

Each part of the chart corresponds to a case of the function. Figure 2 relates the parts of the chart to the cases of the function.

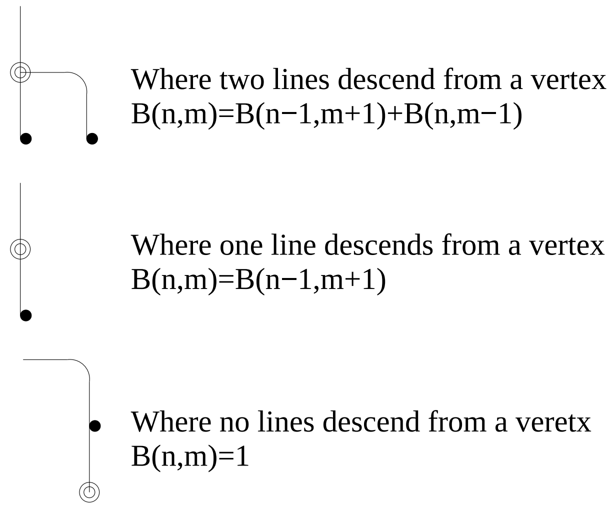


Figure 2: Relationship between the chart and cases of the function

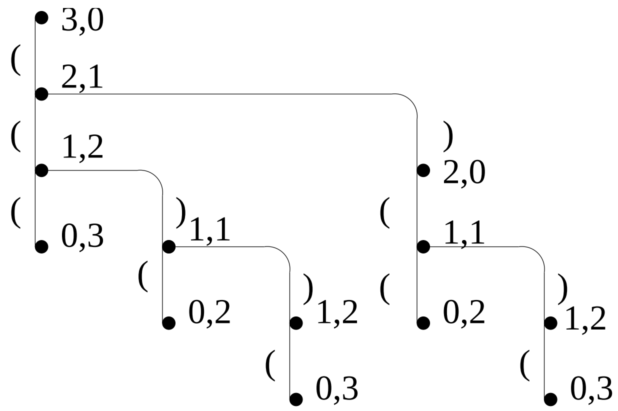


Figure 3: Parameters of $B_{3,0}$ at each vertex

The parameters of the function $B_{n,m}$ take on different values at different points in the recursive descent. Figure 3 shows the parameters at each stage for $B_{3,0}$.

Parameter n represents the number of left parentheses that can be used from that point onward. Parameter m represents the number of additional right parentheses needed to balance the number of left parentheses already used. Considered constructively, m represents the number of right parentheses that may be written at that point. When a left parenthesis is written, n is reduced and m is increased. When a right parenthesis is written, m is reduced.

Of course, once the function is defined, it is freed of any necessary tie to parentheses.

If we say it is possible for any recursive function to be simple, then this function is simple and perhaps more fundamental than the closed form. The closed form is simpler to write. However, while the notation for "$2n$ choose n" is simple, it implies more complex ideas. The closed form has multiplication and division operations. While the comparisons in the recursive functions are obvious and explicitly shown, there are also comparisons implied in any evaluation of the closed form.

Assuming that it is not possible to do algebra with the recursive function, it seems less useful than the closed form. However, it is possible to do substitutions. For example, $B_{4,1}$ can be restated as $B_{3,2} + B_{4,0}$, and vice versa. Substitution could be used to define the function differently, but the way the function was defined above seems simple and it fits well with the parentheses chart.

$B_{n,0}$ is far less efficient computationally than the closed form. This will be developed in the Appendix.

3.3 The Generalization

This function is a generalization of Catalan numbers. The standard Catalan number $C_n = B_{n,0}$. Table 1 also includes some of the others:

	M=0	1	2	3	4	5
N=0	1	1	1	1	1	1
1	1	2	3	4	5	6
2	2	5	9	14	20	27
3	5	14	28	48	75	110
4	14	42	90	165	275	429
5	42	132	297	572	1001	1638
6	132	429	1001	2002	3640	6188
7	429	1430	3432	7072	13260	23256
8	1430	4862	11934	25194	48450	87210

Table 1: Generalized Catalan Numbers $B_{n,m}$ for $n \in [0, 8]$, $m \in [0, 5]$.

3.4 The Structure

The structure can be expressed as:

$$C_n = B_{n-3,3} + 2C_{n-1}$$

or as

$$C_n = B_{n-3,3} + 2B_{n-1,0}$$

The chart for C_n can be characterized as having a left lobe and two equal right lobes. The right lobes are equal both in structure and value. They are also each equal to C_{n-1} in structure and value. Figures 4, 5, and 6 show the structure.

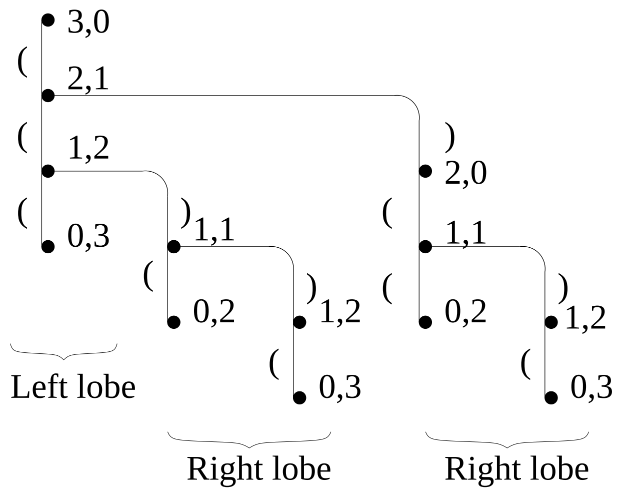


Figure 4: Structure of C_n

In Figure 5, the numeric parameters are replaced by symbolic parameters in terms of n and m. The chart "grows" from the bottom as n increases. The three lobes will always have the values $B_{n-3,3}$, $B_{n-2,1}$, and $B_{n-2,1}$. These can be put in correspondence to the ways legal strings of parentheses may start: (((, (( ), ( ) (. This is a basis of a partition of any set of legal strings.

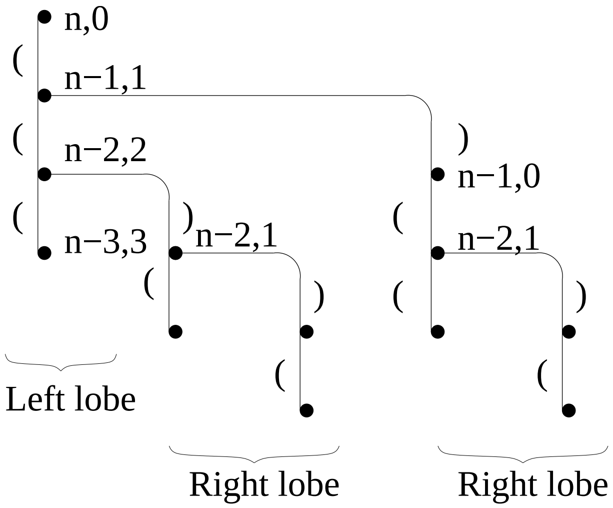


Figure 5: Structure of C_n contd.

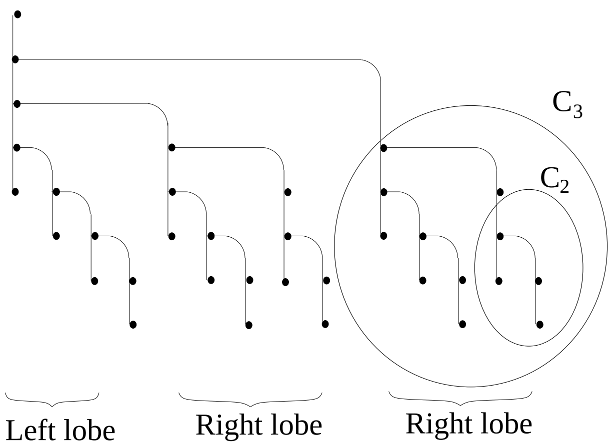


Figure 6: Structure of C_4 or $B_{4,0}$

Figure 6 emphasizes the nested repetitions of structure. Note that C_3 (or C_{n-1}) is found twice and C_2 (or C_{n-2}) is found four times.

The left lobe is different. It starts out smaller than either right lobe and then becomes larger, perhaps approaching the sum of the two right lobes as n gets large. The value of the left lobe is $B_{n-3,3}$. Here's a table of the first few values:

$\mathbf{n}$	3	4	5	6	7	8
$\mathbf{B_{n-3,3}}$	1	4	14	48	165	572

Table 2: Values of the left lobe $B_{n-3,3}$ for $n \in [3, 8]$.

These values were recognized by the On-Line Encyclopedia of Integer Sequences as Sequence A002057, named the Fourth Convolution of Catalan Numbers [5]. This sequence is not pursued here.

4 Further Work

1. What are applications or interpretations of the generalized Catalan numbers?

2. There is doubtless something inherent in the problem that is reflected in the structure, but it is not obvious what. The structure looks natural in terms of the chart, but the chart is just one picture of one interpretation. Why not two lobes? Four? Why any?

3. What is the precise behavior of the size of the left lobe?

4. Is $B_{n-3,3}$ the Fourth Convolution of Catalan numbers?

5 Conclusion

Consideration of the set of legal strings of n pairs of parentheses exposes a structure of this set and of Catalan numbers. The rules for construction of legal strings of parentheses can be recast from a general statement of principles to particular statements of all the cases. This restatement can be expressed as a chart showing all of the cases.

Examination of the chart shows the structure of the sets of strings. Given that the count of legal strings is known to be the Catalan number, the chart exposes a simple and easily understood structure of Catalan numbers. Interpreting the chart as a graph, a recursive function $B_{n,m}$ counts the leaves of the graph (a tree) and therefore calculates the Catalan number.

Taken together, the chart and the function provide a useful tool for gaining an intuitive understanding of an important combinatorial number. Developing the function would be a good problem for students studying recursive functions.

The function $B_{n,m}$ is interesting in its own right. First, it is remarkably simple, using only addition, subtraction, and comparison. It should probably should be considered more fundamental than the closed form which additionally uses multiplication, division, and factorials. Second, the function $B_{n,m}$ has two parameters and is thus a generalization of Catalan numbers.

6 Appendix. Computational Complexity and Efficiency.

The closed form for calculating C_n is clearly more efficient than the recursive $B_{n,0}$. However, examining complexity and efficiency can further illuminate the structure of parenthetical strings and Catalan numbers. The complexity of the closed form is linear in n while that of $B_{n,0}$ is exponential.

This section will only treat $B_{n,0}$ to facilitate comparison with the closed form. The term "C_n" is used here to denote the number, not the method of calculating it.

6.1 Comparison with the closed form.

Even without a precise expression for the complexity of $B_{n,m}$, it is possible to reason about complexity and do some measurements of it. The reasoning goes like this: 1) The complexity of $B_{n,0}$ is greater than the number C_n. 2) C_n is greater than the complexity of the closed form. 3) Therefore the complexity of $B_{n,0}$ is greater than that of the closed form. (It is much greater.)

The unit counted for the closed form is the number of multiplications. After canceling common factors in the numerator and the denominator, the closed form can be expressed as $(2n)(2n-1)...(2n-(n+2))$, calling for $n-2$ multiplications, here called $f(n)$.

The unit counted for $B_{n,0}$ is the number of executions of the function. In many architectures these two measures would not be commensurate. However, the sizes of the complexity numbers dominate any difference. Using C_n as a complexity number, the expression $(2n)(2n-1)...(2n-(n+2))$ expands to a degree $n-1$ polynomial in n, here called $g(n)$.

It can be seen that $f(n)$ is little-oh of $g(n)$ since $\lim_{n\to\infty} f(n)/g(n) = 0$. In other words, f(n) grows more slowly than g(n). In this case it grows much more slowly [8].

The fact that the complexity of $B_{n,0}$ is greater than the number C_n is clear from the chart. The chart has C_n leaves, each contributing 1 to the number of executions. In addition there are many intermediate nodes above the leaves, so that the sum of all executions is greater than C_n. All this demonstrates that the computational complexity of the closed form is little-oh of the complexity of $B_{n,0}$.

A numeric measurement of $B_{n,0}$ is shown in Table 3. (The algorithm based on $B_{n,m}$ can be instrumented to count executions by the appropriate placement of "+1" in the cases of the function.)

$\mathbf{n}$	3	4	5	6	7	8
$\mathbf{n-2}$	1	2	3	4	5	6
$\mathbf{B_{n,0}}$	13	36	106	328	1034	3485

Table 3: Complexity of the closed form vs. $B_{n,0}$.

6.2 Complexity of different implementations of $B_{n,m}$.

6.2.1 "Bottom-up" implementation of a recursive function.

Due to the highly repetitive structure of $B_{n,m}$, results toward the bottom of the chart are recalculated many times over. To justify this, consider that the tree gets much wider than it is high. For example, at $n = 8$ the number of leaves is $C_n = 1430$. The longest path from the root to a leaf is $2n-1$. This shows that many of the computations are towards the bottom.

Blass and Gurevich use the term "bottom-up" to describe the use of precalculated results to avoid many recalculations [1]. As an example, the following fragment of pseudo-code expresses the $B_{n,m}$ as an algorithm. It avoids recalculation of $B_{n,m}$ for $m, n \in [0, 3]$.

The values of T are from Table 1. Note that the cases are not disjoint. The order of execution resolves ambiguity.

```
var T = new Array ([1,1,1,1], [1,2,3,4], [2,5,9,14], [5,14,28,48]);
function B(n,m) {
    if ((n<4)&&(m<4))     return (T[n][m]);
    if ((n>0)&&(m>0))     return (B(n-1, m+1) + B(n, m-1));
    if ((n>0)&&(m==0))    return (B(n-1, m+1));
    if (n==0)             return (1);
}
```

Table 4 shows measured complexity for this version.

n	3	4	5	6	7	8
top-down	13	36	106	328	1054	3485
bottom-up	1	2	5	13	52	212

Table 4: Complexity of top-down vs. bottom up evaluation of $B_{n,0}$.

6.2.2 Parallel Processing.

The structure of $B_{n,m}$ presents both obstacles and opportunities for parallelization. The word "executions" will be used here the way "processes" and "threads" are often used.

Dividing the work.

It is easy to divide the function into parts to run on separate processors. Consider placing a horizontal line on a drawing of the chart such as Figure 4. Horizontal lines can be drawn at various levels. The point at which the new line intersects a vertical line marks a place where a separate process can consist of all the executions below the intersection. The level of the horizontal line would determine the number of parts. This method would be suitable for a multi-processor with few processors.

Another approach uses the fact that the second case calls for two child evaluations of the function. One of these could be sent to another processor. This would lead to many requests for processors at large n.

Latency.

Latency is another important factor in parallelization. "Latency" is used here to mean the time to initiate and terminate an execution, including passing parameters and returning results. Since the amount of processing in the function is small, latency would be very important if the function were distributed over many processors.

A Single Instruction Multiple Data (SIMD) machine with many processors and low latency would be good here. It would also take advantage of the fact that each execution of the algorithm would use the same small program. However, in general the structure of the function would limit its use on machines with large numbers of processors unless latency was very small.

Inter-process communication.

Since there would be no peer-to-peer communication among executions, an execution would never be interrupted and suspended in the middle of processing. Network contention and overhead would both benefit from this characteristic of $B_{n,m}$. Of course, there is much passing of parameters and results. This contributes to latency, as developed above, and would be a significant use of resources.

References

[1] A. Blass and Y. Gurevich, Algorithms vs. Machines, *Bulletin of the European Association for Theoretical Computer Sciences*, Number 77, pp.96-118, June 2002.

[2] R. Graham, D. Knuth, and O. Patashnik, *Concrete Mathematics: A foundation for Computer Science, 1 ed.*, Reading, Mass.: Addison-Wesley, 1988.

[3] P. Hilton and J. Pedersen, Catalan Numbers, Their Generalization, and Their Uses, *The Mathematical Intellegencer*, Volume 13, Number 2, pp.64-75, 1991.

[4] P. Hilton, D. Holton, and J. Pedersen, *Mathematical Vistas: From a Room with Many Windows*, New York, Springer-Verlag, 2002.

[5] N. Sloane, *On-Line Encyclopedia of Integer Sequences*, published electronically at `http://www.research.att.com/~njas/sequences/Seis.html`, 2002.

[6] R. Stanley, *Enumerative Combinatorics Volume 2*, New York, Cambridge University Press, 1999.

[7] D. Stanton and D. White, *Constructive Combinatorics*, New York, Springer-Verlag, 1986.

[8] G. Thomas and R. Finney, *Calculus and Analytic Geometry, 8 ed.*, Reading, Mass., Addison-Wesley, 1992, Reprinted with corrections, April 1993.

Border Array on Bounded Alphabet [1]

Jean-Pierre Duval[2], Thierry Lecroq[2], Arnaud Lefebvre[3]

[2]LIFAR – ABISS, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France
[3]UMR 6037 – ABISS, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France

e-mail: Jean-Pierre.Duval, Thierry.Lecroq, Arnaud.Lefebvre @univ-rouen.fr

Abstract. In this article we present an on-line linear time algorithm, to check if an integer array f is a border array of some string x built on a bounded size alphabet, which is simplest that the one given in [2]. Furthermore if f is a border array we are able to build, on-line and in linear time, a string x on a minimal size alphabet for which f is the border array.

Key words: String algorithms, border array

1 Introduction

A border u of a string x is a prefix and a suffix of x such that $u \neq x$. The computation of the borders of each prefix of a string x is strongly related to the string matching problem: given a string x, find the first or, more generally, all its occurrences in a longest string y. The border array of x is better known as the "failure function" introduced in [4] (see also [1]). Recently, in [2] a method is presented to check if an integer array f is a border array for some string x. The authors first give an on-line linear time algorithm to verify if f is a border array on an unbounded size alphabet. Then they give a more complex algorithm that works on a bounded size alphabet. Here we present a more simple algorithm for this case. Furthermore if f is a border array we are able to build, on-line and in linear time, a string x on a minimal size alphabet for which f is the border array. The resulting algorithm is elegant and integrates three parts: the checking on an unbounded alphabet, the checking on a bounded size alphabet and the design of the corresponding string if f is a border array. The first two parts can work independently.

The remaining of this article is organized as follows. The next section introduces basic notions and notations on strings and results from [2]. Section 3 presents our new algorithm together with its correctness proof. Finally we give our conclusions in Sect. 4.

2 Background and basic string definitions

A *string* is a sequence of zero or more symbols from an alphabet Σ; the string with zero symbols is denoted by ε. The set of all strings over the alphabet Σ is denoted

[1]This work was partially supported by a NATO grant PST.CLG.977017.

by Σ^*. We consider an alphabet of size s; for $1 \le i \le s$, $\sigma[i]$ denotes the i-th symbol of Σ. A string x of length n is represented by $x[1..n]$, where $x[i] \in \Sigma$ for $1 \le i \le n$. A string u is a *prefix* of x if $x = uw$ for $w \in \Sigma^*$. Similarly, u is a *suffix* of x if $x = wu$ for $w \in \Sigma^*$. A string u is a *border* of x if u is a prefix and a suffix of x and $u \ne x$.

Let $f[1..n]$ be an integer array such that $f[i] < i$ for $1 \le i \le n$. For $1 \le i \le n$, we define $f^1[i] = f[i]$ and for $f[i] > 0$, $f^\ell[i] = f[f^{\ell-1}[i]]$. We use the following notations:

- $L(f, i-1) = (f[i-1], f^2[i-1], \ldots, f^m[i-1] = 0)$;
- $C(f, i) = (1 + f[i-1], 1 + f^2[i-1], \ldots, 1 + f^m[i-1])$ where $f^m[i-1] = 0$.

Note that $L(f, 1) = (0)$ and that $C(f, 1)$ is not defined.

A border u of $x[1..i]$ with $i > 0$ has one of the two following forms:

- $u = \varepsilon$;
- $u = x[1..j]x[j+1]$ with $j + 1 < i$ and where $x[1..j]$ is a border of $x[1..i-1]$ and $x[i] = x[j+1]$.

For $1 \le i \le n$ we denote by $\beta_x[i]$ the length of the longest border of $x[1..i]$. The array $\beta_x[1..n]$ is said to be the border array of the string x.

The lengths of the different borders of $x[1..i-1]$ are given by the decreasing sequence

$$L(\beta_x, i-1) = (\beta_x[i-1], \beta_x^2[i-1], \ldots, \beta_x^m[i-1])$$

where $\beta_x^m[i-1] = 0$ i.e. it is the length of the longest border $\beta_x[i-1]$ followed by the lengths of the borders of this longest border $L(\beta_x, \beta_x[i-1])$.

For $i \ge 2$, we say that an integer $j+1$ is candidate to be the length of the longest border of $x[1..i]$ if $x[1..j]$ is a border of $x[1..i-1]$. In other words, for $i \ge 2$, saying that $j+1$ is candidate means that $j \in L(\beta_x, i-1)$. The decreasing sequence of candidates for the length of the longest border of $x[1..i]$ is

$$C(\beta_x, i) = (1 + \beta_x[i-1], 1 + \beta_x^2[i-1], \ldots, 1 + \beta_x^m[i-1])$$

where $\beta_x^m[i-1] = 0$.

We say that an array $f[1..n]$ is a *valid border array*, or simply that it is *valid* if and only if it is the border array of at least one string x of length n.

The longest border of $x[1]$ is necessarily the empty word, thus $\beta_x[1] = 0$. The length $\beta_x[i]$ of the longest border of $x[1..i]$, if it is not empty, is taken among the candidates $C(\beta_x, i)$. Thus we have a first necessary condition for an array $f[1..n]$ to be valid:

$$NC_1\text{: } f[1] = 0 \text{ and for } 2 \le i \le n, f[i] \in \{0\} + C(f, i)\,.$$

If $x[1..i]$ has the empty word for only border then we have $\beta_x[i] = 0$.

If $x[1..i]$ has a non-empty border, the length of the longest border verifies

- $\beta_x[i] = \max\{j + 1 \mid j \in L(\beta_x, i-1) \text{ and } x[i] = x[j+1]\}$, or equivalently
- $\beta_x[i] = \max\{j + 1 \mid j + 1 \in C(\beta_x, i) \text{ and } x[i] = x[j+1]\}$.

The length $j+1$ of the longest border of $x[1..i]$ is the first candidate in the list $C(\beta_x, i)$ for which $x[j+1] = x[i]$ if it exists, otherwise the longest border has length 0. This is the basis of the computation of the function β_x known as a "failure function" given in [4].

Saying that $j+1$ is the largest candidate for which $x[j+1] = x[i]$ implies that this is not true for any candidate $j'+1$ larger than $j+1$, which imposes that $x[1..j+1]$ cannot be a border of $x[1..j'+1]$ for a candidate $j'+1$ larger than $j+1$. In other words, $\beta_x[j'+1]$ is different from $j+1$ for any candidate $j'+1$ larger than $j+1$.

This is thus a second necessary condition for an array f to be valid:

NC_2: for $i \geq 2$ and for every $j'+1 \in C(f, i)$ with $j'+1 > f[i]$
we have $f[j'+1] \neq f[i]$.

Theorem 2.2 in [2] states that conditions NC_1 and NC_2 form a sufficient condition for f to be a valid border array. The authors give, for any valid array f, thus satisfying conditions NC_1 and NC_2, the computation of a string x such that $f = \beta_x$, without any restriction on the alphabet size. They give a simple linear time algorithm (Theorem 2.3) to test if an array f satisfies conditions NC_1 and NC_2, on a unbounded size alphabet. They give a more complex algorithm in the case of a bounded size alphabet. Here we present a more simple algorithm which determines in linear time, for a given array $f[1..n]$, for i from 1 to n, the minimum size of an alphabet necessary to build a string $x[1..i]$ which border array is $f[1..i]$.

3 New algorithm

We propose, in this section, a linear time algorithm, which determines, for an array $f[1..n]$ and an alphabet size s given as input:

1 – validity: if $f[1..n]$ is a valid border array for at least one string $z[1..n]$. This point is essentially the same as in [2];

2 – alphabet: up to which index it is possible to build a string which border array is f using an alphabet of size s;

3 – string: a string x, on a minimal size alphabet, which border array is f.

Point 1 is independent from the other two points. Point 2 can work without the other two points, in particular when one assumes that the array f is valid and does not want to build a corresponding string. Point 3 uses point 2.

The algorithm BABA (for Border Array on Bounded Alphabet) is given figure 1.

We now state our main result.

Theorem 1 *When applied to an integer array $f[1..n]$ and an alphabet size s:*

- *The algorithm BABA runs in time $\Theta(n)$.*
- *If the array f given as input of the algorithm BABA is a valid border array at index $i-1$ but not at index i, the algorithm stops and returns "f invalid at index i". The lines* {**alphabet**} *and* {**string**} *can be deleted without changing this result.*

```
BABA(f, n, s)
if f[1] ≠ 0                                                  ▷ validity
  then return f invalid at index 1                           ▷ validity
k[1] ← 1                                                     ▷ alphabet
x[1] ← σ[1]                                                  ▷ string
for i ← 2 to n
    do if f[i] = 0
          then k[i] ← 1 + k[f[i − 1] + 1]                    ▷ alphabet
               if k[i] > s                                   ▷ alphabet
                 then return s exceeded at index i           ▷ alphabet
               x[i] ← σ[k[i]]                                ▷ string
          else j ← f[i − 1]                                  ▷ validity
               while j + 1 > f[i] and f[j + 1] ≠ f[i]        ▷ validity
                   do j ← f[j]                               ▷ validity
               if j + 1 ≠ f[i]                               ▷ validity
                 then return f invalid at index i            ▷ validity
               k[i] ← k[f[i − 1] + 1]                        ▷ alphabet
               x[i] ← x[f[i]]                                ▷ string
return x
```

Figure 1: Algorithm BABA

- *If there exists a string for which $f[1..i-1]$ is the border array and there is none at index i with an alphabet of size s, the algorithm BABA stops and returns "s exceeded at index i". Lines* {**string**} *can be deleted without changing this result. If the array f is valid, lines* {**validity**} *can also be deleted.*

- *As long as $f[i..1]$ is valid, the algorithm BABA builds a string $x[1..i]$ on a minimal size alphabet for the border array $f[1..i]$. Lines* {**validity**} *can be deleted without changing the construction of the string. It is clear that if f is invalid, it is not the border array of the string which is built by the algorithm.*

Before giving the proof of the previous theorem we first give a definition and establish some intermediate results.

Definition 1 *Given a string $x[1..n]$ and its border array β_x, we denote by $A(x,i)$ the set of symbols that extend the prefix $x[1..i-1]$ and its borders, in x: $A(x,i) = \{x[i]\} \cup \{x[j+1] \mid j+1 \in C(\beta_x, i)\}$.*

Figure 2 gives a description of $L(\beta_x, i-1)$, $C(\beta_x, i)$ and $A(x,i)$.

Lemma 1 *For every string $x[1..i]$ we have*

1. *$\{x[j+1] \mid j+1 \in C(\beta_x, i)\} = A(x, \beta_x[i]+1)$;*
2. *If $\beta_x[i] \neq 0$ then $x[i] = x[\beta_x[i]]$, $\beta_x[i] \in C(\beta_x, i)$ and $A(x,i) = A(x, \beta_x[i-1]+1)$.*
3. *If $\beta_x[i] = 0$ then $\beta_x[i] \notin C(\beta_x, i)$ and $A(x,i) = \{x[i]\} \cup A(x, \beta_x[i-1]+1)$.*

Proof:

1. Immediate;

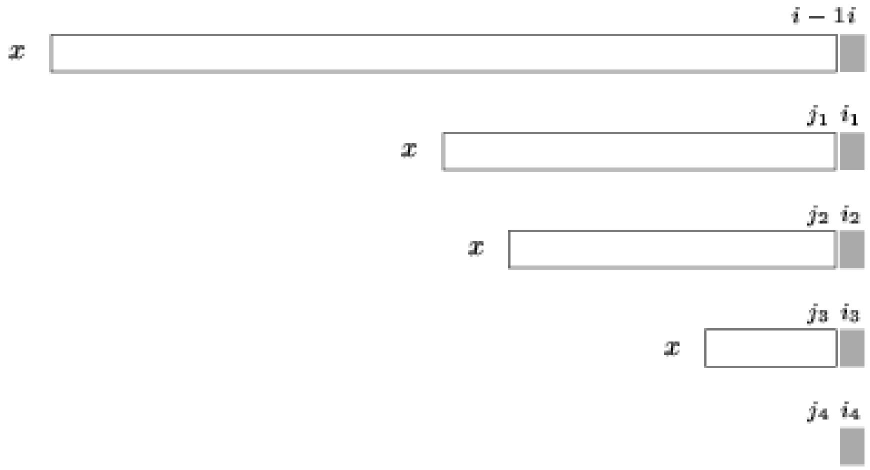


Figure 2: If for $1 \leq \ell \leq 4$, $j_\ell = \beta_x^\ell[i-1]$, $j_4 = \beta_x^4[i-1] = 0$, $i_\ell = 1 + \beta^\ell[i-1]$, then $L(\beta_x, i-1) = (j_1, j_2, j_3, j_4 = 0)$, $C(\beta_x, i) = (i_1, i_2, i_3, i_4 = 1)$ and $A(x,i)$ is the set which is composed of the gray symbols.

2. If $\beta_x[i] \neq 0$ then $\beta_x[i]$ is a candidate of $C(\beta_x, i)$. Concerning the index of the longest border we have $x[i] = x[\beta_x[i]]$, $\beta_x[i]$ is a candidate in $C(\beta_x, i)$, $x[i]$ is in $A(x, \beta_x[i-1]+1)$;
3. $\beta_x[i] = 0$ implies that there exists no candidate $j+1 \in C(\beta_x, i)$ such that $x[i] = x[j+1]$.

$\square$

Corollary 1 *Let $x[1..n]$ be a string and $k[1..n]$ the array computed by the algorithm BABA with the input $f = \beta_x$ ignoring the* {**validity**} *and* {**string**} *lines. Then, for $1 \leq i \leq n$ we have $k[i] = card A(x,i)$.*

Proof: The proof of the corollary immediately follows from the algorithm BABA and properties 2 and 3 of lemma 1. $\square$

Corollary 2 *For every string x which border array is f, the minimal cardinality of an alphabet necessary to build each prefix $x[1..i]$ is greater or equal to* $\max\{k[1], k[2], \ldots, k[i]\}$ *where $k[1..n]$ is the array computed by the algorithm BABA with the input $f = \beta_x$, ignoring lines* {**validity**} *and* {**string**}.

Proof: All the symbols of $A(x,j)$ for $1 \leq j \leq i$ are symbols of the string $x[1..i]$. Thus the cardinality is greater or equal to the cardinality of each $A(x,j)$. $\square$

Proposition 1 *Assume that array $f[1..n]$ is valid. The string x build by the algorithm BABA satisfies the following properties:*

1. *For $1 \leq i \leq n$, $\beta_{x[1..i]} = f[1..i]$ and $A(x,i) = \{\sigma[1], \sigma[2], \ldots, \sigma[k[i]]\}$;*
2. *The cardinality of the alphabet for each prefix $x[1..i]$ is equal to*

$$\max\{k[1], k[2], \ldots, k[i]\};$$

3. *The border array β_x of the string x is equal to f.*

Proof:

- We show the point 1 by induction on i. For $i = 1$: $f[1] = 0$, $\beta_{x[1..1]} = f[1..1]$ and $A(x, 1) = \{x[1]\} = \{\sigma[1]\}$. The property holds at index 1.

  Assume that the property holds up to index $i-1$, then we have $A(x, f[i-1]+1) = \{\sigma[1], \sigma[2], \ldots, \sigma[k[f[i-1]+1]]\}$ (since $f[i-1] < i-1$ thus $f[i-1]+1 \leq i-1$) and $\beta_{x[1..i-1]} = f[1..i-1]$.

  If $f[i] \neq 0$ then since $f[1..i-1] = \beta_{x[1..i-1]}$ and f satisfies conditions NC_1 and NC_2 at index i, $f[i]$ is the largest candidate j of $C(f, i)$ such that $x[j] = x[f[i]]$. Thus, by setting $x[i] \leftarrow x[f[i]]$ we get $\beta_x[i] = f[i]$, $k[i] = k[f[i-1]+1]$ and $A(x, i) = A(x, f[i-1]+1) = \{\sigma[1], \sigma[2], \ldots, \sigma[k[i]]\}$.

  If $f[i] = 0$ then $k[i] = 1 + k[f[i-1]+1]$ and $x[i] \leftarrow \sigma[k[i]]$ does not belong to $A(x, f[i-1]+1)$ thus $\beta_x[i] = 0$, $A(x, \beta_x[i-1]+1) = \{\sigma[1], \sigma[2], \ldots, \sigma[k[f[i-1]+1]]\}$, $A(x, i) = \{\sigma[k[i]]\} \cup A(x, \beta_x[i-1]+1) = \{\sigma[1], \sigma[2], \ldots, \sigma[k[i]]\}$.

  The property holds for i in both cases.

- Properties 2 and 3 are immediate consequences of property 1.

□

Proposition 2 *Let $f[1..n]$ be an integer array.*

1. *The algorithm BABA returns "f invalid at index i" if and only if $f[1..i-1]$ is valid and $f[1..i]$ is not;*

2. *The array $f[1..i-1]$ is the border array of the string $x[1..i-1]$ which is built by the algorithm BABA.*

Proof: From proposition 1, as long as $f[1..i]$ is valid, it is the border array of the string $x[1..i]$ which is built by the algorithm BABA which establishes the point 2.

If the algorithm BABA stops at index $i = 1$ and returns "f invalid at index 1", it means that $f[1] \neq 0$ thus $f[1..i]$ is invalid (note that this case cannot happen if the condition $f[i] < i$ is fulfilled).

Now assume that at the beginning of iteration i we have: $z[1..i-1]$ is a string which border array is $f[1..i-1]$ and z can be extended with a symbol $z[i]$ for which $\beta_z[i] = f[i]$.

We have $z[i] = z[f[i]]$, and $\beta_z[i] = f[i]$ is the largest candidate $j'+1 \in C(\beta_z, i) = (1+\beta_z[i-1], 1+\beta_z^2[i-1], \ldots, 1+\beta_z^m[i-1])$, such that $z[j'+1] = z[i]$ thus it is the largest for which $z[j'+1] = z[f[j]]$.

The three lines {**validity**} of the algorithm BABA reviews in decreasing order the candidates $j+1$ of $C(\beta_z, i)$.

- If the algorithm exits the while loop with $j+1 > f[i]$ and $f[j+1] = f[i]$, it means that $j+1$ is a candidate larger that $f[i]$ for which $\beta_z[j+1] = f[i]$ thus $z[j+1] = z[f[i]]$ which contradicts the fact that $j'+1$ is the largest candidate such that $z[j'+1] = z[f[i]]$. This contradicts the assumption that the string $z[1..i-1]$ can be extended and that $f[1..i]$ is valid.

- If the algorithm exits the while loop with $j+1 < f[i]$, it means that no candidate $j'+1$ equal to $f[i]$ were found. This contradicts the fact that $f[i] = \beta_z[i]$ and that $f[1..i]$ is valid.

In both cases, no string $z[1..i-1]$, which border array is $f[1..i-1]$, can be extended, then the algorithm returns "f invalid at index i".

If $f[1..i]$ is valid then the algorithm does not stop at this index.

Assume now that at the beginning of iteration i we have: $z[1..i-1]$ is a string which border array is $f[1..i-1]$ and the while loop exits at index i with $j+1 = f[i]$.

Let us set $z[i] = z[f[i]]$. Then $f[i] = j+1$ is a candidate of $C(\beta_z, i)$ for which $z[j+1] = z[i]$ thus $z[1..j+1]$ is a border of $z[1..i]$. Assume that $z[1..j+1]$ is not the longest border of $z[1..i]$. Let $j'+1$ be the smallest candidate which is larger than $j+1$ and such that $z[1..j'+1]$ is a border of $z[1..i]$. Then $z[1..j+1]$ is the longest border of $z[1..j'+1]$ and we have $f[j'+1] = f[i]$ which means that the loop should have stop with this test and with $j+1 > f[i]$. This is a contradiction.

Thus the algorithm BABA runs as long as $f[1..i]$ is valid, it stops at index i and returns "f invalid at index i" if and only if f is valid up to index $i-1$ and is not at index i. □

The proof of Theorem 1 becomes then immediate.

Proof:[of Theorem 1] The point 1 (linearity of the algorithm BABA) comes from [4]. The other two points follow from propositions 1 and 2. □

Figures 3 and 4 show two examples.

i	1	2	3	4	5	6	7	8	9	10	11	12	symbols	candidates	valid
$x[i]$	a	b	a	a	b	a	b	a	a	b	a				
$f[i]$	0	0	1	1	2	3	2	3	4	5	6	?			
$k[i]$	1	2	1	2	2	1	2	1	2	2	1				
						a	b	a	a	b	a		b	7	YES
									a	b	a		a	4	YES
											a		b	2	NO
											ε		a	1	NO
													c	0	YES IF $s > 2$

Figure 3: The array $f[1..11]$ is a valid border array. The string $x[1..11]$ is the smallest string for which $f[1..11]$ is a valid border array. Then $x[1..11] =$ abaababaaba has borders abaaba, aba, a and ε of respective lengths 6, 3, 1 and 0 ($L(f, 11) = (6, 3, 1, 0)$). Thus the candidates for $f[12]$ are 7, 4, 2 and 1 ($C(f, 12) = (7, 4, 2, 1)$) together with 0 which is always a potential candidate. The values 7 and 4 are valid candidates. The value 2 is not valid since $f[7] = 2$ and 1 is not valid because $f[4] = 1$. The value 0 is a valid candidate if $s > 2$ because then $k[12]$ would be equal to $1 + k[f[12-1]+1] = 3$.

4 Conclusions

We presented in this article an elegant algorithm that verify, on-line and in linear time, if an integer array f is a border array of some string on a bounded size alphabet.

i	1	2	3	4	5	6	7	8	9	10	11	12	symbols	candidates	valid
$x[i]$	a	a	b	a	a	c	a	a	b	a	a				
$f[i]$	0	1	0	1	2	0	1	2	3	4	5	?			
$k[i]$	1	1	2	1	1	3	1	1	2	1	1				
							a	a	b	a	a		c	6	YES
										a	a		b	3	YES
											a		a	2	YES
											ε		a	1	NO
													d	0	YES IF $s > 3$

Figure 4: The array $f[1..11]$ is a valid border array. The string $x[1..11]$ is the smallest string for which $f[1..11]$ is a valid border array. Then $x[1..11] =$ `aabaacaabaa` has borders `aabaa`, `aa`, `a` and ε of respective lengths 5, 2, 1 and 0 ($L(f, 11) = (5, 2, 1, 0)$). Thus the candidates for $f[12]$ are 6, 3, 2 and 1 ($C(f, 12) = (6, 3, 2, 1)$) together with 0 which is always a potential candidate. The values 6, 3 and 2 are valid candidates. The value 1 is not valid since $f[2] = 1$. The value 0 is a valid candidate if $s > 3$ because then $k[12]$ would be equal to $1 + k[f[12 - 1] + 1] = 4$.

In the case where f is a border array, we are also capable to build a string x, on a minimal size alphabet for which f is the border array.

After studying the case of the "failure function" of the Morris and Pratt string matching algorithm, it is natural to ask the question if this work can be extended to the "failure function" of the Knuth, Morris and Pratt string matching algorithm [3].

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1974.

[2] F. Franěk, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun and L. Yang, Verifying a border array in linear time, *J. Comb. Math. Comb. Comput.* **42** (2002) to appear.

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

[4] J. H. Morris, Jr and V. R. Pratt, A linear pattern-matching algorithm, Report 40, University of California, Berkeley, 1970.

A Note on Crochemore's Repetitions Algorithm a Fast Space-Efficient Approach[1]

František Franěk[1], W. F. Smyth[1,2], and Xiangdong Xiao[1]

[1] Algorithms Research Group, Department of Computing & Software
McMaster University, Hamilton, Ontario, Canada L8S 4K1
(www.cas.mcmaster.ca/cas/research/groups.html)
[2] School of Computing, Curtin University, GPO Box U-1987
Perth WA 6845, Australia

e-mail: franek@mcmaster.ca, smyth@mcmaster.ca

Abstract. The space requirement of Crochemore's repetitions algorithm is generally estimated to be about $20MN$ bytes of memory, where N is the length of the input string and M the number of bytes required to store the integer N. The same algorithm can also be used in other contexts, for instance to compute the suffix tree of the input string in $O(N \log N)$ time for the purpose of data compression. In such contexts the large space requirement of the algorithm is a significant drawback. There are of course several newer space-efficient algorithms with the same time complexity that can compute suffix trees or arrays. However, in actual implementations, these algorithms may not be faster than Crochemore's. Therefore, we consider it interesting enough to describe a new approach based on the same mathematical principles and observations that were put forth in Crochemore's original paper, but whose space requirement is $10MN$ bytes. Additional advantages of the approach are the ease with which it can be implemented in C/C++ (as we have done) and the apparent speed of such an implementation in comparison to other implementations of the original algorithm.

1 Introduction

Crochemore's algorithm [C81] computes all the repetitions in a finite string $\boldsymbol{x}$ of length N in $O(N \log N)$ time. The algorithm in fact computes rather more and can be used, for instance, to compute the suffix tree of $\boldsymbol{x}$, hence possibly as a tool for expressing $\boldsymbol{x}$ in a compressed form. In such contexts the space requirement becomes as important as the time complexity. It appears that known implementations of Crochemore's algorithm require at least $20MN$ bytes of memory for the task of refining the equivalence classes alone, where M is the number of bytes required to store the integer N.

Here we present a different implementation based on the mathematical properties and observations of [C81] and thus having the same time complexity $O(N \log N)$ as the original algorithm. However, the new data structures used for the representation

[1]Supported in part by grants from the Natural Sciences & Engineering Research Council of Canada.

of classes and for the execution of the refinement process allow the space requirement to be substantially reduced.

There are several newer space-efficient algorithms to compute suffix trees or arrays (notably [U92], [MM93]) of the same worst-case complexity as Crochemore's. The motivation for our investigation of a space-efficient implementation of the classical Crochemore's algorithm that may be competetive with these newer algorithms stems from the fact that the actual implementations of these algorithms may not in fact be any faster.

A large memory saving comes from the fact that our algorithm requires storage for only N classes at any given time, rather than $2N$ as in the original algorithm. This alone brings the space requirement down to $15MN$. Of course there is some extra processing related to this reduction in space, but it does not affect the time complexity, and in fact it appears that in practice our implementation runs a good deal faster than the standard implementation proposed in [C81]. A further $5MN$ space reduction is achieved by smart utilization of the space:

- allowing space to be shared by data structures, as in memory multiplexing — for example, if a queue empties faster than a stack grows, then they can share the same memory segment;
- spreading one data structure across several others, as in memory virtualization.

Taken together, these "tricks" bring the space requirement down to $10MN$.

Additional advantages of this approach are the ease with which it can be implemented in C/C++ (as we have done) and, as remarked above, its apparent speed in comparison to other implementations of the original algorithm.

In this paper we do not due to space limitations provide any detailed computer instructions, but we try to give a high-level description of our approach, so that the reader can understand how the space savings are achieved.

In our discussion below we assume that the reader is familiar with both Crochemore's algorithm and its mathematical foundation. We make the usual assumption required for Crochemore's algorithm that the alphabet is ordered; therefore we are able to assume further that the classes corresponding to the first level ($p = 1$) can be computed in $O(N \log N)$ time.

For better comprehension, we present the algorithm in two stages. The first stage, FSX15 (with space requirement $15MN$ bytes), exhibits all important procedural and control aspects of our algorithm without the complications of memory multiplexing and virtualization. Then the second stage, FSX10, incorporates the changes required by memory multiplexing and virtualization to reduce the space requirement to $10MN$. Finally, we present some rough results of computer runs that compare the time and space requirements of our approach with those of a standard implementation of Crochemore's algorithm.

2 Data Structures for FSX15

Recall that for each $p = 1, 2, \ldots$, Crochemore's algorithm acts on a given string $\boldsymbol{x} = \boldsymbol{x}[1..N]$ to compute equivalence classes $\{i_1, i_2, \ldots, i_r\}$, where for every $1 \le j < h \le r$,

$$\boldsymbol{x}[i_j..i_j{+}p{-}1] = \boldsymbol{x}[i_h..i_h{+}p{-}1].$$

The positions i_j in each class are maintained in increasing sequence: $i_j < i_{j+1}$, $1 \leq j < r$. At each step of the algorithm, each class $\mathcal{C}_p$ that is not a singleton is decomposed into a ***family*** of subclasses $\mathcal{C}_{p+1,s}$; of these subclasses, the one of largest cardinality is called ***big***, the others are ***small***. A straightforward approach to this decomposition would require order N^2 time in the worst case, but Crochemore's algorithm reduces this time requirement by carrying out the decomposition from p to $p+1$ only with respect to the small classes identified at step p. Since each position can belong to a small class only $O(\log N)$ times, it follows that the total time requirement is $O(N \log N)$. As remarked in the introduction, we may assume that the classes corresponding to $p = 1$ have initially been computed in $O(N \log N)$ time. Note that the version of Crochemore's algorithm discussed here does not explicitly compute repetitions; we will be interested only in reducing each of the equivalence classes to a singleton.

We will use an integer array of size N to represent the classes computed at step p. We have several requirements:

- we need to keep the elements of the classes in ascending order;
- we need an efficient way to delete any element (so that we need to represent each class as a doubly-linked list);
- we need an efficient way to insert a new element at the end of a class (and hence we need a link to the last element of the class);
- we need efficient access to the size of a class;
- we need efficient access to a class (and hence we need a link to the first element of the class);
- last but not least, we need an efficient way to determine to which class a given element belongs.

To satisfy all these requirements, we use six integer arrays of size N:

- `CNext[1..N]` emulates forward links in the doubly-linked list. Thus `CNext`$[i] = j > i$ means that j is the next element (position) in the class that i belongs to. If there is no position $j > i$ in the class, then `CNext`$[i] =$ `null`.
- `CPrev[1..N]` emulates backward links in the doubly-linked list. Thus `CPrev`$[i] = j < i$ means that j is the previous element (position) in the class that i belongs to. If there is no position $j < i$ in the class, then `CPrev`$[i] =$ `null`.
- `CMember[1..N]` keeps track of membership. Thus `CMember`$[i] = k$ means that element i belongs to the class with index k ($i \in c_k$), while `CMember`$[i] =$ `null` means that at this moment i is not member of any class.
- `CStart[1..N]` keeps links to the starting (smallest) element in each class. Thus `CStart`$[k] = i$ means that the class c_k starts with the element i, while `CStart`$[k] =$ `null` means that at this moment the class c_k is empty.

- `CEnd[1..N]` keeps links to the final (largest) element in each class. Thus $\texttt{CEnd}[k] = i$ means that the class c_k ends with the element i; the value of $\texttt{CEnd}[k]$ is meaningful only when $\texttt{CStart}[k] \neq \texttt{null}$.

- `CSize[1..N]` records the size of each class. Thus $\texttt{CSize}[k] = r$ means that the class c_k contains r elements; the value of $\texttt{CSize}[k]$ is meaningful only when $\texttt{CStart}[k] \neq \texttt{null}$.

Suppose that there exists a class $c_3 = \{4, 5, 8, 12\}$, indicating that the substrings of length 3 beginning at positions $4, 5, 8, 12$ of $\boldsymbol{x}$ are all equal. Then c_3 would be represented as follows:

$$\texttt{CNext}[4] = 5,\ \texttt{CNext}[5] = 8,\ \texttt{CNext}[8] = 12, \texttt{CNext}[12] = \texttt{null};$$
$$\texttt{CPrev}[12] = 8,\ \texttt{CPrev}[8] = 5,\ \texttt{CPrev}[5] = 4,\ \texttt{CPrev}[4] = \texttt{null};$$
$$\texttt{CMember}[4] = \texttt{CMember}[5] = \texttt{CMember}[8] = \texttt{CMember}[12] = 3;$$
$$\texttt{CStart}[3] = 4; \quad \texttt{CEnd}[3] = 12; \quad \texttt{CSize}[3] = 4.$$

We need to track the empty classes, and for that we need a simple integer stack of size N, `CEmptyStack`, that holds the indexes of the empty (and hence available) classes. This stack, as well as all other list structures used by Crochemore's algorithm, is implemented as an array that requires MN bytes of storage. Such an approach saves time by allowing all space allocation to take place only once, as part of program initialization. We introduce two operations on the stack, $\boldsymbol{CEmptyStackPop()}$ that removes the top element from the stack and returns it, and $\boldsymbol{CEmptyStackPush(i)}$ that inserts the element i at the top of the stack.

We shall process classes from one refinement level p to the next level $p+1$ by moving the elements from one class to another, one element at a time. We view the classes as permanent containers and distribute the elements among them, so that at any given moment we need at most N classes. This means that the configuration of classes at level p is destroyed the moment we move a single element. But, as we shall see, we do not really need to keep the old level intact if we preserve an essential "snapshot" of it before we start tinkering with it.

What we need to know about level p will be preserved in two queues, `SElQueue` and `SCQueue`. `SElQueue` contains all the elements in small classes in level p, organized so that the elements from the same small class are grouped together in the queue and stored in ascending order. `SCQueue` contains the first element from each small class, thus enabling us to identify in `SElQueue` the start of each new class. Therefore, when these queues are created, we must be careful to process the small classes of level p in the same order for both of them. For instance, if level p had three small classes,

$$c_3 = \{2, 4, 5, 8\},\ c_0 = \{3, 6, 7, 11\},\ c_5 = \{12, 15\},$$

`SElQueue` could contain $2, 4, 5, 8, 3, 6, 7, 11, 12, 15$ in that order, while the corresponding `SCQueue` would contain $2, 3, 12$. The order of the classes (c_3 followed by c_0 followed by c_5) is not important; what is important that the same order is used in order to create `SElQueue` and `SCQueue`. After the two queues have been created, we do not need level p any more and we can start modifying it. Of course we suppose that we have available the usual queue operations:

- ***SElQueueHead*()** (remove the first element from the queue and return it);
- ***SElQueueInsert*($\boldsymbol{i}$)** (insert the element i at the end of the queue);
- ***SElQueueInit*()** (initialize the queue to empty).

Analogous operations are available also for `SCQueue`.

When refining class c_k in level p using an element i from class $c_{k'}$, we might need to move element i–1 from c_k to a new or an existing class. To manage this processing, we keep an auxiliary array of size N, `Refine[1..N]`. Initially, when we start using the class $c_{k'}$ for refinement, all entries in `Refine[ ]` are `null`. If a new class c_h is created in level p+1 by moving $i-1$ out of class c_k and into c_h as its first element, we set `Refine`$[k] \leftarrow h$. If later on we move another element from c_k as a result of refinement by the same class $c_{k'}$, we use the value `Refine`$[k]$ to tell us where to move it to. This requires that when we start refining by a new class, we have to restore `Refine[ ]` to its original `null` state. Since we cannot afford to traverse the whole array `Refine[ ]` without destroying the $O(N \log N)$ time complexity, we need to store a record of which positions in `Refine[ ]` were previously given a non-`null` value. For this we make use of a simple stack, `RefStack`: every assignment to `Refine`$[k]$ causes the index k to be pushed onto the stack `RefStack`. As before, we assume that we have available the usual stack operations ***RefStackPop*()** and ***RefStackPush*($\boldsymbol{i}$)**.

Since after completing the refinement of the classes in level p, we must determine the small classes in level p+1, we therefore need to maintain throughout the refinement process certain families of classes (to be more precise, families of class indexes). As noted above, a family consists of the classes in level $p + 1$ that were formed by refinement of the same class in level p. A family may or may not include the original class from level p itself (it may completely disappear if we remove all its elements during the refinement). We need an efficient way to insert a new class in a family (the order is not important), an efficient way to delete a class from a family, and finally an efficient way to determine to what family (if any) a class belongs. These facilities can be made available by representing the families as doubly-linked lists implemented using arrays, just as we did previously with the classes themselves. In this case, however, the `Size[ ]` and `End[ ]` arrays are not required, so we can get by with only four arrays, as follows:

- `FNext[1..N]` emulates the forward links (as in `CNext[ ]`).
- `FPrev[1..N]` emulates the backward links (as in `CPrev[ ]`).
- `FMember[1..N]` keeps track of membership (as in `CMember[ ]`). Whenever `FMember`$[i]$ = `null`, it means that c_i is not a member of any family.
- `FStart[1..N]` gives the first class in each family (as in `CStart[ ]`).

Note that classes in families do not need to be maintained in numerical order, as was true earlier of positions in classes.

To summarize, in order to implement Crochemore's algorithm, it is sufficient to allocate 15 arrays, each of which provides storage space for exactly N integers of length M, thus altogether $15MN$ bytes of storage: `CNext`, `CPrev`, `CMember`, `CStart`, `CEnd`, `CSize`, `CEmptyStack`, `SElQueue`, `SCQueue`, `RefStack`, `Refine`, `FStart`, `FNext`, `FPrev`, and `FMember`.

3 Data Structures for FSX10

As the first step in reducing the space complexity further, we are going to eliminate the `CSize[ ]` and `CEnd[ ]` arrays. For the very first element, say s, in a class, `CPrev[`s`]`$=$ `null`, while for the very last element, say e, `CNext[`e`]`$=$ `null`. But we have another way to discern the beginning of the class (`CStart[ ]`), so that `CPrev[ ]` becomes superfluous. Thus we can store `CPrev[`s`]`$\leftarrow e$, a direct link to the end of the class. This yields an efficient means to discern the end of the class, and so we can store in `CNext[`e`]` the size of the class. Hence `CPrev[CStart[`j`]]` takes on the role of `CEnd[`j`]`, while `CNext[CPrev[CStart[`j`]]]` takes on the role of `CSize[`j`]`. This is straightforward and the code need only be slightly modified to accommodate it. All we have to do is make sure that when inserting or deleting an element in or from a class, we update properly the end link and the size. When traversing a class, we have to make sure that we properly recognize the end (we cannot rely on the `null` value to stop us as in `FSX15`). We have in fact "virtualized" the memory for `CEnd[ ]` and `CSize[ ]`, and so reduced the space complexity to $13MN$.

When we take an element from `SElQueue` and use it for the purpose of refinement, at most one new class is created and thus at most one location of `Refine[ ]` is updated. This simple observation allows `RefStack` and `SElQueue` to share the same memory segment, as long as we make sure that `RefStack` grows from left to right, while the queue is always right justified in the memory segment. The changes in the code required to accommodate this are not very great — all we have to do is to determine before filling `SElQueue` what position we have to start with. In essence, we have "multiplexed" the same memory segment and brought the space complexity down to $12MN$.

The number of elements in `SCQueue` is the same as the number of small classes, which is less than or equal to the number of non-empty classes; thus the size of `SCQueue` plus the size of `CEmptyStack` at any given moment is at most N. This simple observation allows `CEmptyStack` and `SCQueue` to share the same memory segment, as long as we make sure that `CEmptyStack` is growing from left to right, while the queue is always right justified in the memory segment. Again, as above, the changes in the code required to accommodate this are not major. We again have "multiplexed" the same memory segment and brought the space complexity down to $11MN$.

The final memory saving comes from the fact that `FPrev[ ]` for the very first class in a family and `FNext[ ]` for the very last class in the same family are set to `null` and hence redundant for the same reasons as described above for `CPrev[ ]` and `CNext[ ]`. We can thus "virtualize" the memory for the array `Refine[ ]`. We will have to index it in reverse and we will use all the unused slots in `FStart[ ]` (i.e. slots with indexes $>$ `FStartTop`) as well as the unnecessary `FNext[ ]` slots. The formula is rather simple. Instead of storing r in `Refine[`i`]`, we will use

```
SetRefine(i,r)
j←N-(i+1)
if FStartTop = null OR j > FStartTop then
   FStart[j]←r
else
   FNext[FPrev[FStart[j]]]←r
end SetRefine
```

and instead of fetching a value from `Refine[i]` we will use

```
integer GetRefine(i)
j←N-(i+1)
if FStartTop = null OR j > FStartTop then
   return FStart[j]
else
   return FNext[FPrev[FStart[j]]]
end GetRefine
```

The modification of the code is more complex in this case, since we have to track the ends of the family lists as we do for class lists; more importantly, when a new family is created, we have to save the `Refine[ ]` value stored in that so-far-unused slot k that now is going to be occupied by the start link of the family list, and store k at the end of the list instead. This "virtualization" of the memory for `Refine[ ]` brings the space complexity down to the final value of $10MN$.

4 Informative Experimental Results

To estimate the effect of our space reduction on time requirement, we have implemented two versions of Crochemore's algorithm:

- a naïve array-based version, FSX20, that executes Crochemore's algorithm using 20 arrays each of length N words:
- a version of FSX10 that requires 10 arrays each of length N words.

Thus both of these implementations are word-based: assuming a word-length of 32 bits, the value of M is actually fixed at 4.

We expect that FSX20 will execute Crochemore's algorithm about as fast as it can be executed, but at the cost of requiring exactly $20N$ words of storage. A version that implemented standard list-processing techniques rather than arrays to handle the queues, stacks and lists required by Crochemore's algorithm would generally require less storage: $11N$ words for arrays plus a variable amount up to $13N$ for the list structures. However, as a result of the time required for dynamic space allocation, such a version would certainly run several times slower than FSX20.

We must remark at this point that the experiments performed have only an informative value, for we conducted them without controlling many aspects depending on the platform (as memory caching, virtual memory system paging etc.), nor did we perform a proper statistical evaluation to control for other factors not depending on the platform (load on the machine, implementation biases etc.) Thus, we really do not claim any significant conclusions for the actual algorithms whose implementations were tested.

We have run FSX20 and FSX10 against a variety of long strings (up to 3.8 million bytes): long Fibonacci strings, files from the Calgary corpus, and others. The results indicate that FSX10 seems to require 20-30% more time than FSX20, in most cases a small price to pay for a 52% reduction in space.

References

[C01] Calgary Corpus

`http://links.uwaterloo.ca/calgary.corpus.html`

[C81] Maxime Crochemore, **An optimal algorithm for computing the repetitions in a word**, *IPL 12-5* (1981) 244-250.

[MM93] Udi Manber & Gene W. Myers, **Suffix arrays: a new method for on-line string searches**, *SIAM J. Comput. 22-5* (1993) 935-948.

[U92] Esko Ukkonen, **Constructing suffix trees on-line in linear time**, *Proc. IFIP 92*, vol. I (1992) 484-492.

A Bit-Vector Algorithm for Computing Levenshtein and Damerau Edit Distances[1]

Heikki Hyyrö

Department of Computer and Information Sciences
33014 University of Tampere
Finland

e-mail: `Heikki.Hyyro@uta.fi`

Abstract. The edit distance between strings A and B is defined as the minimum number of edit operations needed in converting A into B or vice versa. The Levenshtein edit distance allows three types of operations: an insertion, a deletion or a substitution of a character. The Damerau edit distance allows the previous three plus in addition a transposition between two adjacent characters. To our best knowledge the best current practical algorithms for computing these edit distances run in time $O(dm)$ and $O(\sigma + \lceil m/w \rceil n)$, where d is the edit distance between the two strings, m and n are their lengths ($m \leq n$), w is the computer word size and σ is the size of the alphabet. In this paper we present an algorithm that runs in time $O(\sigma + \lceil d/w \rceil m)$. The structure of the algorithm is such, that in practice it is mostly suitable for testing whether the edit distance between two strings is within some pre-determined error threshold. We also present some initial test results with thresholded edit distance computation. In them our algorithm works faster than the original algorithm of Myers.

Key words: Levenshtein edit distance, Damerau edit distance, bit-parallelism, approximate string matching

1 Introduction

The desire to measure the similarity between two strings may arise in many applications, like for example computational biology and spelling correction. A common way to achieve this is to compute the edit distance between the strings. Throughout the paper we will assume that A is a string of length m and B is a string of length n, and that $m \leq n$. The edit distance $ed(A, B)$ between strings A and B is defined as the minimum number of edit operations needed in converting A into B or vice versa. In this paper we concentrate on two typical edit distances: the Levenshtein edit distance [Lev66] and the Damerau edit distance [Dam64]. The Levenshtein edit distance allows three edit operations, which are inserting, deleting or substituting a character (Figures 1a, 1b and 1c). In addition to these three, the Damerau edit distance allows also transposing two permanently adjacent characters (Figure 1d). When edit

[1]This work was supported by the Academy of Finland and Tampere Graduate School in Information Science and Engineering

distance is used, strings A and B are deemed similar iff their edit distance is small enough, that is iff $ed(A, B) \leq k$, where k is some pre-determined error threshold. A related problem is that of approximate string matching, which is typically defined as follows: let *pat* be a string of length m and *text* a (much longer) string of length n. The task of approximate string matching is then to find all such indices j, for which exists such $h \geq 0$ that $ed(pat, text[j-h..j]) \leq k$.

The oldest, but most flexible in terms of permitting different edit operations and/or edit operation costs, algorithms for computing edit distance (for example [WF74]) are based on dynamic programming and run in time $O(mn)$. Ukkonen [Ukk85a] has later proposed two $O(dm)$ methods, and Myers [Mye86] an $O(n + d^2)$ method. The latter is based on using a suffix tree and is not viewed as being practical (e.g. [Ste94]). With fairly little modifications these methods can also be used in computing the Damerau edit distance without affecting the asymptotic run times.

The methodology of using so-called "bit-parallelism" in developing fast and practical algorithms has recently become popular in the field of string matching. Wu and Manber [WM92] presented an $O(d\lceil m/w \rceil n)$ bit-parallel algorithm for Levenshtein edit distance -based approximate string matching, and in [Nav01] it was modified to compute both Levenshtein and Damerau edit distance. The run time remained the same. Then Baeza-Yates and Navarro presented a method, which enables an $O(\lceil dm/w \rceil n)$ algorithm for the Levenshtein edit distance. Currently this algorithm has not been extended for the Damerau edit distance. Finally Myers [Mye99] has presented an $O(\lceil m/w \rceil n)$ algorithm for approximate string matching under the Levenshtein edit distance. In [Hyy01] the algorithm was extended for computing the Damerau edit distance.

In this paper we will present an initial study on combining one of the $O(dm)$ edit distance algorithms of Ukkonen [Ukk85a] with the bit-parallel algorithm of Myers [Mye99] to obtain a faster algorithm. We begin by reviewing these underlying algorithms in the next section.

2 Preliminaries

In the following discussion let A be a string of length m and B a string of length n. We also use the notation $A[u]$ to denote the uth character of A and the notation $A[u..v]$ to denote the substring of A, which begins from its uth character and ends at its vth character. The superscript R denotes the reverse string: for example if A = "ABC", then A^R = "CBA". For bit operations we use the following notation: '&' denotes bitwise"and", '|' denotes bitwise "or", '$\wedge$' denotes bitwise "xor", '$\sim$' denotes bit complementation, and '<<' and '>>' denote shifting the bit-vector left and right, respectively, using zero filling in both directions. We refer to the ith bit of the bit vector V as $V[i]$. Bit-positions are assumed to grow from right to left, and we use superscript to denote bit-repetition. As en example let $V = 1001110$ be a bit vector. Then $V[1] = V[5] = V[6] = 0$, $V[2] = V[3] = V[4] = V[7] = 1$, and we could also write $V = 10^2 1^3 0$.

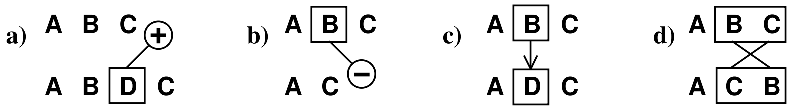


Figure 1: Four different edit operations. Figure a) shows inserting character 'D' between the last two characters of the string "ABC", which results in the string "ABCD". Figure b) shows deleting the character "B", which results in the string "AC". Figure c) shows substituting the character 'B' with the character 'D', which results in the string "ADC". Figure d) shows transposing the characters 'B' and 'C', which results in the string "ACB". Transposition is allowed only between such characters that were adjacent already in the original string.

2.1 Dynamic programming

Computing edit distance is a problem that seems to be most naturally solved with dynamic programming. The value $ed(A, B)$ can be computed by filling an $(m+1) \times (n+1)$ dynamic programming matrix D, in which the cell $D[i, j]$ contains the value $ed(A[1..i], B[1..j])$. The following well-known Recurrence 1 gives the rule for filling D when the Levenshtein edit distance is used.

Recurrence 1

$$D[i,0] = i, D[0,j] = j.$$
$$D[i,j] = \begin{cases} D[i-1,j-1], \text{ if } A[i] = B[j]. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), \text{ if } A[i] \neq B[j]. \end{cases}$$

The recurrence allows the cells with $i > 0$ and $j > 0$ to be filled in any such order, that the cell values $D[i-1,j]$, $D[i-1,j-1]$ and $D[i,j-1]$ are known at the time the cell $D[i,j]$ is filled. A common way is to use column-wise filling, where each column is filled from top to bottom (Figure 2). The Damerau edit distance can be computed otherwise identically as the Levenshtein edit distance, but using Recurrence 2 [Hyy01] instead in filling the dynamic programming matrix.

Recurrence 2

$$D[i,0] = i, D[0,j] = j.$$
$$D[i,j] = \begin{cases} D[i-1,j-1], & \text{if } A[i] = B[j]. \\ D[i-1,j-1], & \text{if } A[i-1..i] = B^R[j-1..j] \\ & \quad \text{and } D[i-1,j-1] > D[i-2,j-2]. \\ 1 + \min(D[i-1,j-1], D[i-1,j], D[i,j-1]), & \text{otherwise.} \end{cases}$$

As the basic dynamic programming scheme fills $(m+1)(n+1)$ cells and filling each cell takes a constant number of operations, the algorithm has a run time $O(nm)$. The following two properties hold in the dynamic programming matrix [Ukk85a, Ukk85b]:

-The diagonal property: $D[i,j] - D[i-1,j-1] = 0$ or 1.
-The adjacency property: $D[i,j] - D[i,j-1] = -1, 0,$ or 1, and
$D[i,j]D[i-1,j] = -1, 0,$ or 1.

Even though these rules were initially presented with the Levenshtein edit distance, they can easily be seen to apply also with the Damerau edit distance.

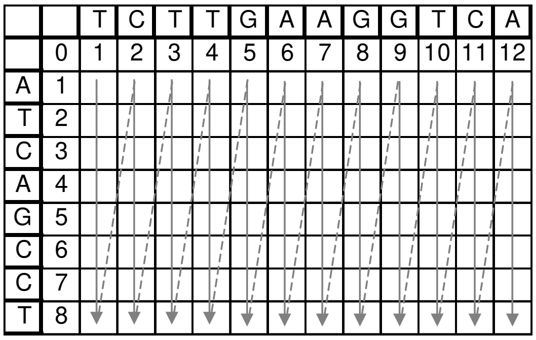


Figure 2: An example of the column-wise filling order for the dynamic programming table of strings "ATCAGCCT" and "TCTTGAAGGTCA".

2.2 Using bit-parallelism

Myers [Mye99] presented an $O(\lceil m/w \rceil n)$ algorithm for approximate string matching under the Levenshtein edit distance. Later in [Hyy01] the algorithm was slightly modified and extended for the Damerau edit distance. Originally these algorithms were designed for approximate string matching, but they can easily be modified to compute edit distance. The algorithms process the jth column of the dynamic programming matrix in $O(\lceil m/w \rceil)$ time by using bit-parallelism. This is done by using delta encoding in the matrix: instead of explicitly computing the values $D[i,j]$ for $i = 1..m$ and $j = 1..n$, the following length-m binary valued delta vectors are computed for $j = 1..n$:

-The vertical positive delta vector: $VP_j[i] = 1$ iff $D[i,j] - D[i-1,j] = 1$.
-The vertical negative delta vector: $VN_j[i] = 1$ iff $D[i,j] - D[i-1,j] = -1$.
-The horizontal positive delta vector: $HP_j[i] = 1$ iff $D[i,j] - D[i,j-1] = 1$.
-The horizontal negative delta vector: $HN_j[i] = 1$ iff $D[i,j] - D[i,j-1] = -1$.

When the values for these delta vectors are known for the $(j-1)$th column, they can be computed for the jth column in an efficient manner when the following match vector is available for each character λ.

-The match vector PM_λ : $PM_\lambda[i] = 1$ iff $A[i] = \lambda$.

For simplicity we use the notion $PM_j = PM_{B[j]}$ for the rest of the paper. It is straightforward to compute the pattern match vectors in $O(\sigma + m)$ time. In the following we assume that these vectors have already been computed and are readily available.

The delta vectors enable the value $ed(A, B[1..j])$ to be explicitly calculated for $j = 1, 2, \ldots, n$: $ed(A, B[1..j]) = ed(A, B[1..j-1]) + 1$ iff $HP_j[m] = 1$, $ed(A, B[1..j]) = ed(A, B[1..j-1]) - 1$ iff $HN_j[m] = 1$, and $ed(A, B[1..j]) = ed(A, B[1..j-1])$ otherwise. Thus after all n columns are processed, the value $ed(A, B[1..n]) = ed(A, B)$ is known. Figures 3 and 4 show the algorithms based on [Hyy01] for computing the jth column when $m \leq w$, that is, when each vector can be represented by a single bit-vector. Both algorithms are modified to compute edit distance. The algorithm in Figure 3 is for the Levenshtein edit distance, and the algorithm in Figure 4 is for the Damerau edit distance. Both algorithms involve a constant number of operations, and thus compute the delta vectors for the jth column in $O(1)$ time. In this paper we do not separately discuss the case $m > w$. As each required operation for a length-m bit vector can be simulated in $O(\lceil m/w \rceil)$ time using $\lceil m/w \rceil$ length-w bit vectors, the general runtime of the algorithms is $O(\lceil m/w \rceil)$ for each column. This results in a total time of $O(\lceil m/w \rceil n)$ over all n columns in computing $ed(A, B)$.

Computing the jth column (Levenshtein distance)

1. $D0_j \leftarrow (((PM_j \ \& \ VP_{j-1}) + VP_{j-1}) \ \wedge \ VP_{j-1}) \mid PM_j \mid VN_{j-1}$
2. $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
3. $HN_j \leftarrow D0_j \ \& \ VP_{j-1}$
4. **If** $HP_j \ \& \ 10^{m-1} \neq 0$ **Then** $D[m, j] \leftarrow D[m, j] + 1$
5. **If** $HN_j \ \& \ 10^{m-1} \neq 0$ **Then** $D[m, j] \leftarrow D[m, j] - 1$
6. $VP_j \leftarrow (HN_j << 1) \mid \sim (D0_j \mid (HP_j << 1)) \mid 1$
7. $VN \leftarrow D0_j \ \& \ (HP_j << 1)$

Figure 3: Computation of the jth column using a modification of the $D0_j$-based version of the algorithm of Myers (for the case $m \leq w$).

Computing the jth column (Damerau distance)

1. $D0_j \leftarrow (((\sim D0_{j-1}) \ \& \ PM_j) << 1) \ \& \ PM_{j-1}$
2. $D0_j \leftarrow D0_j \mid (((PM_j \ \& \ VP_{j-1}) + VP_{j-1}) \ \wedge \ VP_{j-1}) \mid PM_j \mid VN_{j-1}$
3. $HP_j \leftarrow VN_{j-1} \mid \sim (D0_j \mid VP_{j-1})$
4. $HN_j \leftarrow D0_j \ \& \ VP_{j-1}$
5. **If** $HP_j \ \& \ 10^{m-1} \neq 0$ **Then** $D[m, j] \leftarrow D[m, j] + 1$
6. **If** $HN_j \ \& \ 10^{m-1} \neq 0$ **Then** $D[m, j] \leftarrow D[m, j] - 1$
7. $VP_j \leftarrow (HN_j << 1) \mid \sim (D0_j \mid (HP_j << 1)) \mid 1$
8. $VN \leftarrow D0_j \ \& \ (HP_j << 1)$

Figure 4: Computation of the jth column using a modification of the $D0_j$-based version of the algorithm of Myers with transposition (for the case $m \leq w$).

2.3 Filling only a necessary portion of the matrix

Ukkonen [Ukk85a] presented a method to try to cut down the area of the dynamic programming matrix that is filled. By a q-diagonal we refer to the diagonal, which consists of the cells $D[i, j]$ for which $j - i = q$. From the diagonal and adjacency properties Ukkonen concluded that if $ed(A, B) \leq t$ and $m \leq n$, then it is sufficient to fill only the cells in the diagonals $-\lfloor (t - n + m)/2 \rfloor, -\lfloor (t - n + m)/2 \rfloor + 1, \ldots, \lfloor (t + n - m)/2 \rfloor$ of the dynamic programming matrix. All the other cell values can be assumed to have an infinite value without affecting correct computation of the value $D[m, n] = ed(A, B)$. He used this rule by beginning with $t = (n - m) + 1$ and filling the above-mentioned diagonal interval of the dynamic programming matrix. If the result is $D[m,n] > t$, t is doubled. Eventually $D[m,n] \leq t$, and in this case it is known that $D[m,n] = ed(A,B)$. The run time of this procedure is dominated by the computation involving the last value of t. As this value is $< 2 \times ed(A,B)$ and with each value of t the computation takes $O(t \times \min(m,n))$ time, the overall run time is $O(ed(A,B) \times \min(m,n))$.

In addition Ukkonen proposed a dynamic "cutoff" method to improve the practical performance of the diagonal restriction method. Assume that column-wise order is used in filling the cells $D[i,j]$ inside the required diagonals $-\lfloor (t-n+m)/2 \rfloor, -\lfloor (t-n+m)/2 \rfloor + 1, \ldots, \lfloor (t+n-m)/2 \rfloor$. Let r_u hold the diagonal number of the upmost and r_l the diagonal number of the lowest cell that was deemed to have to be filled in the jth column. Then due to the diagonal property we can try to shrink the diagonal region by decrementing r_u as long as $D[r_u, j] > t$ and incrementing r_l as long as $D[r_l, j] > t$. Then at the $(j+1)$th column it is enough to fill the cells in the diagonals $r_l \ldots r_u$. If $r_l > r_u$ the diagonal region vanishes and it is known that $ed(A,B) > t$.

This method of "guessing" a starting limit t for the edit distance and then doubling it if necessary is not really practical for actual edit distance computation. Even though the asymptotic run time is good, it involves large constant factor whenever $ed(A,B)$ is large. But the method works well in practice in thresholded edit distance computation, as then one can immediately set $t = k$ and only a single pass is needed.

3 Our Method

In this section we present a bit-parallel version of the diagonal restriction scheme of Ukkonen, which was briefly discussed in Section 2. In the following we concentrate on the case where the computer word size w is large enough to cover the required diagonal region. Let l_v denote the length of the delta vectors. Then our assumption means that $w \geq l_v = \min(m, \lfloor (t-n+m)/2 \rfloor + \lfloor (t+n-m)/2 \rfloor + 1)$. Note that in this case each of the pattern match vectors PM_λ may have to be encoded with more than one bit vector: If $m > w$, then PM_λ consists of $\lceil m/w \rceil$ bit vectors.

3.1 Diagonal tiling

The basic idea is to mimic the diagonal restriction method of Ukkonen by tiling the vertical delta vectors diagonally instead of horizontally (Figure 5a). We achieve this by modifying slightly the way the vertical delta vectors VP_j and VN_j are used: Before processing the jth column the vertical vectors VP_{j-1} and VN_{j-1} are shifted one step up (to the right in terms of the bit vector) (Figure 5b). When the vertical vectors are shifted up, their new lowest bit-values $VP_j[l_v]$ and $VN_j[l_v]$ are not explicitly known. This turns out not to be a problem. From the diagonal and adjacency properties we can see that the only situation which could be troublesome is if we would incorrectly have a value $VN_j[l_v] = 1$. This is impossible, because it can happen only if $D0_j$ has an "extra" set bit at position $l_v + 1$ and $HP_j[l_v] = 1$, and these two conditions cannot simultaneously be true.

In addition to the obvious way of first computing VP_j and VN_j in normal fashion and then shifting them up (to the right) when processing the $(j+1)$th column, we propose also a second option. It can be seen that essentially the same shifting effect

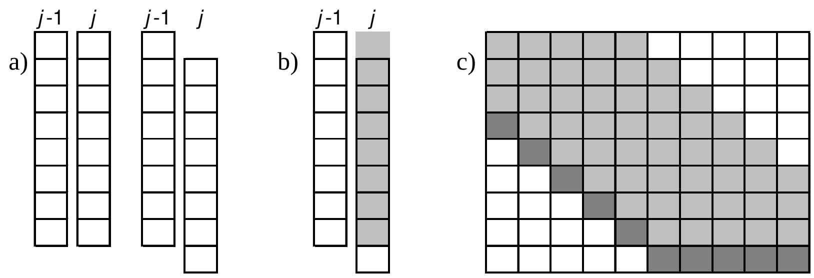


Figure 5: a) Horizontal tiling (left) and diagonal tiling (right). b) The figure shows how the diagonal step aligns the $(j-1)$th column vector one step above the jth column vector. c) The digure depicts in gray the region of diagonals, which are filled according to Ukkonen's rule. The cells on the lower boundary are in darker tone.

can be achieved already when the vectors VP_j and VN_j are computed by making the following changes on the last two lines of the algorithms in Figures 3 and 4:

-The diagonal zero delta vector $D0_j$ is shifted one step to the right on the second last line.

-The left shifts of the horizontal delta vectors are removed.

-The OR-operation of VP_j with 1 is removed.

This second alternative uses less bit operations, but the choice between the two may depend on other practical issues. For example if several bit vectors have to be used in encoding $D0_j$, the column-wise top-to-bottom order may make it more difficult to shift $D0_j$ up than shifting both VP_j and VN_j down.

We also modify the way some cell values are explicitly maintained. We choose to calculate the values along the lower boundary of the filled area of the dynamic programming matrix (Figure 5c). For two diagonally consecutive cells $D[i-1, j-1]$ and $D[i, j]$ along the diagonal part of the boundary this means setting $D[i, j] = D[i-1, j-1]$ if $D0_j[l_v] = 1$, and $D[i, j] = D[i-1, j-1]+1$ otherwise. The horizontal part of the boundary is handled in similar fashion as in the original algorithm of Myers: For horizontally consecutive cells $D[i, j-1]$ and $D[i, j]$ along the horizontal part of the boundary we set $D[i, j] = D[i, j-1] + 1$ if $HP_j[l_v] = 1, D[i, j] = D[i, j-1] - 1$ if $HN_j[l_v] = 1$, and $D[i, j] = D[i, j-1]$ otherwise. Here we assume that the vector length l_v is appropriately decremented as the diagonally shifted vectors would start to protrude below the lower boundary.

Another necessary modification is in the way the pattern match vector PM_j is used. Since we are gradually moving the delta vectors down, the match vector has to be aligned correctly. This is easily achieved in $O(1)$ time by shifting and OR-ing the corresponding at most two match vectors.

The last necessary modifications concern the first line of the algorithm for the Damerau edit distance in Figure 4. First of all the diagonal delta vector $D0_j$ is shifted down (left), which is not necessary when the vectors are tiled diagonally. Because of similar reason the vector PM_{j-1} has to be shifted one step up (to the right). This means that also the value $PM_{j-1}[l_v + 1]$ will have to be present in the match vector PM_{j-1}. We do not deal with this separately, but assume for now on that $l_v + 1 \leq w$ when dealing with the Damerau edit distance. Another option would be to set the last bit separately, which can be done in $O(1)$ time.

Figures 6 and 7 show the algorithms for computing the vectors at the jth column when diagonal tiling is used. We do not show separate versions for the different cases of updating the cell value at the lower boundary. It is done using one of the previously mentioned ways of using $D0_j$ (diagonal stage) or HP_j and HN_j (horizontal stage).

When $l_v \leq w$, each column of the dynamic programming matrix is computed in $O(1)$ time, which results in the total time being $O(\sigma + n)$ including also time for preprocessing the pattern match vectors. In the general case, in which $l_v > w$, each length-l_v vector can be simulated by using $\lceil l_v/w \rceil$ length-w vectors. This can be done in $O(\lceil l_v/w \rceil)$ time per operation, and therefore the algorithm has in general a run time $O(\sigma + \lceil l_v/w \rceil n)$, which is $O(\sigma + ed(A,B) \times n)$ as $l_v = O(ed(A,B))$. The slightly more favourable time complexity of $O(\sigma + ed(A,B) \times m)$ in the general case can be achieved by simply reversing the roles of the strings A and B: We still have that $l_v = O(ed(A,B))$, but now there is m columns instead of n. In this case the cost of preprocessing the match vectors is $O(\sigma + n)$, but the above complexities hold since $n = O(ed(A,B) \times m)$ when $n > m$.

Computing the jth column in diagonal tiling (Levenshtein distance)

1. **Build the correct match vector into PM_j**
2. $D0_j \leftarrow (((PM_j \ \& \ VP_{j-1}) + VP_{j-1}) \ \wedge \ VP_{j-1}) \mid PM_j \mid VN_{j-1}$
3. $HP_j \leftarrow VN_{j-1} \mid \ \sim (D0_j \mid VP_{j-1})$
4. $HN_j \leftarrow D0_j \ \& \ VP_{j-1}$
5. **Update the appropriate cell value at the lower boundary.**
6. $VP_j \leftarrow HN_j \mid \ \sim ((D0_j >> 1) \mid HP_j)$
7. $VN \leftarrow (D0_j >> 1) \ \& \ HP_j$

Figure 6: Computation of the jth column with the Levenshtein edit distance and diagonal tiling (for the case $l_v \leq w$).

Computing the jth column in diagonal tiling (Damerau distance)

1. **Build the correct match vector into PM_j**
2. $D0_j \leftarrow (\sim D0_{j-1}) \ \& \ (PM_j << 1) \ \& \ (PM_{j-1} >> 1)$
3. $D0_j \leftarrow D0_j \mid (((PM_j \ \& \ VP_{j-1}) + VP_{j-1}) \ \wedge \ VP_{j-1}) \mid PM_j \mid VN_{j-1}$
4. $HP_j \leftarrow VN_{j-1} \mid \ \sim (D0_j \mid VP_{j-1})$
5. $HN_j \leftarrow D0_j \ \& \ VP_{j-1}$
6. **Update the appropriate cell value at the lower boundary.**
7. $VP_j \leftarrow HN_j \mid \ \sim ((D0_j >> 1) \mid HP_j)$
8. $VN \leftarrow (D0_j >> 1) \ \& \ HP_j$

Figure 7: Computation of the jth column with the Damerau edit distance and diagonal tiling (for the case $l_v \leq w$).

4 Test Results

In this section we present initial test results for our algorithm in the case of computing the Levenshtein edit distance. We concentrate only on the case where one wants to check whether the edit distance between two strings A and B is below some predetermined error-threshold k. This is because the principle of the algorithm makes it in practice most suitable for this type of use. Therefore all tested algorithms used a scheme similar to the cutoff method briefly discussed in the end of Section 2.3. As a baseline we also show the runtime of using the $O(\lceil m/w \rceil n)$ bit-parallel algorithm of Myers.

The test consisted of repeatedly selecting two substrings in pseudo-random fashion from the DNA-sequence of the baker's yeast, and then testing whether their Levenshtein edit distance is at most k. The computer used in the tests was a 600Mhz Pentium 3 with 256MB RAM and running Microsoft Windows 2000. The code was programmed in C and compiled with Microsoft Visual C++ 6.0 with full optimization. The tested algorithms were:

MYERS: The algorithm of Myers [Mye99] (Section 2.2) modified to compute edit distance. The run time of the algorithm does not depend on the number of errors allowed. The underlying implementation is from the original author.

MYERS(cutoff): The algorithm of Myers using cutoff modified to compute edit distance. The underlying implementation (including the cutoff-mechanism) is from the original author.

UKKA(cutoff): The method of Ukkonen based on filling only a restricted region of diagonals in the dynamic programming matrix and using the cutoff method (Section 2.3).

UKKB(cutoff): . The method of Ukkonen [Ukk85a] based on computing the values in the dynamic programming matrix in increasing order. That is, the method first fills in the cells that get a value 0, then the cells that get a value 1, and so on until the cell $D[m, n]$ gets a value.

OURS(cutoff): Our method of combining the diagonal restriction scheme of Ukkonen with the bit-parallel algorithm of Myers (Section 3). We implemented a similar cutoff method as was used by Hyyrö and Navarro with edit distance computation in their version of the ABNDM algorithm [HN02].

The results ase shown in Figure 8. We tested sequence pairs with lengths 100, 1000 and 10000, and error thresholds of 10%, 20% and 50% of the sequence length (for example $k = 100$, 200 and 500 for the sequence length $m = n = 1000$). It can be seen that in the case of $k = 10$ and $m = 100$ UKKB(cutoff) is the fastest, but in all other tested cases our method becomes the fastest, being 8%-38% faster than the original cutoff method of Myers that is modified to compute edit distance. The good performance of UKKB(cutoff) with a low value of k is not surprising as its expected run time has been shown to be $O(m + k^2)$. [Mye86].

	$m = n = 100$			$m = n = 1000$			$m = n = 10000$		
error limit (%)	10	20	50	10	20	50	10	20	50
UKKA(cutoff)	1,92	5,93	32,6	13,5	52,7	322	13,1	54,9	351
UKKB(cutoff)	1,23	3,02	14,9	6,17	22,9	139	5,57	22,4	146
MYERS(cutoff)	2,46	3,23	4,07	2,47	4,48	15,9	0,71	2,35	13,4
OURS(cutoff)	2,27	2,47	3,32	1,96	3,08	10,5	0,48	1,47	9,03
MYERS	4,24			17,0			14,5		

Figure 8: The results (in seconds) for thresholded edit distance computation between pairs of randomly chosen DNA-sequences from the genome of the baker's yeast. The error threshold is shown as the percentage of the pattern length (tested pattern pairs had equal length). The number of processed sequence pairs was 100000 for $m = n = 100$, 10000 for $m = n = 1000$, and 100 for $m = n = 10000$.

Conclusions and further considerations

In this paper we discussed how bit-parallelism and a diagonal restriction scheme can be combined to achieve an algorithm for computing edit distance, which has an asymtotic run time of $O(\sigma + \lceil d/w \rceil m)$. In practice the algorithm is mostly suitable for checking whether $ed(A, B) \leq k$, where k is a pre-determined error threshold. In this task the initial tests showed our algorithm to be competitive against other tested algorithms [Ukk85a, Mye99], which have run times $O(dm)$ and $O(\sigma + mn/w)$, respectively.

During the preparation of this article we noticed that there seems to be a lack of comprehensive experimental comparison of the relative performance between different algorithms for computing edit distance. Thus we are planning to fill this gap in the near future by composing a fairly comprehensive survey on algorithms for computing edit distance. The survey will also include a more comprehensive test with our algorithm.

We would also like to point out that the algorithm pseudocodes we have shown have not been optimized to remain more clear. Practical implementations could for example avoid shifting the same variable twice and maintain only the needed delta vector values in the memory (the delta vectors in the jth column are only needed when processing the $(j + 1)th$ column).

References

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171–176, 1964.

[HN02] H. Hyyrö and G. Navarro. Faster bit-parallel approximate string matching. In *Proc. 13th Combinatorial Pattern Matching (CPM'2002)*, LNCS 2373, pages 203–224, 2002.

[Hyy01] H. Hyyrö. Explaining and extending the bit-parallel algorithm of Myers. Technical Report A-2001-10, University of Tampere, Finland, 2001.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original in Russian in *Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.*

[Mye86] G. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[Nav01] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265–1312, 2001.

[Ste94] G. A. Stephen. *String Searching Algorithms.* World Scientific, 1994.

[Ukk85a] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[Ukk85b] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[WF74] R. Wagner and M. Fisher. The string to string correction problem. *Journal of the ACM*, 21:168–178, 1974.

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

String Matching with Gaps for Musical Melodic Recognition

Costas S. Iliopoulos[†] and Masahiro Kurokawa[‡]

[†] Algorithm Design Group, Dept of Computer Science,
King's College London, Strand, England, and
School of Computing, Curtin University of Technology,
Perth, Australia
[‡] Algorithm Design Group, Dept of Computer Science,
King's College London, Strand, England

e-mail: `csi@dcs.kcl.ac.uk, kurokawa@dcs.kcl.ac.uk`

Abstract. Here, we have designed and implemented algorithms for string matching with gaps for musical melodic recognition on polyphonic music using bit-wise operations. Music analysts are often concerned with finding occurrences of patterns (motifs), or repetitions of the same pattern, possibly with variations, in a score. An important example of flexibility required in score searching arises from the nature of polyphonic music. Within a certain time span each of the simultaneously-performed voices in a musical composition does not, typically, contain the same number of notes. So 'melodic events' occurring in one voice may be separated from their neighbours in a score by intervening events in other voices. Since we cannot generally rely on voice information being present in the score we need to allow for temporal 'gaps' between events in the matched pattern.

Key words: exact string matching, approximate string matching, gaps, pattern recognition, computer-assisted music analysis, bit-wise operation

1 Introduction

This paper focuses on a set of string pattern-matching problems that arise in musical analysis, and especially in musical information retrieval. Music analysts are often concerned with finding occurrences of patterns, or repetitions of the same pattern, possibly with variations, in a score, while computer scientists often have to perform similar tasks on strings (sequences of symbols from an alphabet). Many objects can be viewed as strings: a text file, for instance, is a sequence of characters from the ASCII alphabet; a DNA code is a sequence of characters from the alphabet A,C,G,T (representing the base proteins which constitute DNA). Similarly, a musical score can

[†] Partially supported by a Marie Curie fellowship, Wellcome and Royal Society grants.

be viewed (at one level) as a string: at a very rudimentary level, the alphabet could simply be the set of notes in the chromatic or diatonic notation, or the set of intervals that appear between notes (e.g. pitch may be represented as MIDI numbers and pitch intervals as number of semitones).

Monophonic music (that is, music in which a single note only sounds at any given time) lends itself well to a one-dimensional string matching approach, and efficient matching algorithms for single-line melody-retrieval have been applied with some success. The polyphonic situation (where several voices or instruments may be performing together, and any number of notes may be sounding at any given time) is more complex, however, because of the temporal interaction between non-simultaneous events in different voices. Where full knowledge about the voicing of the music data (in both the search-pattern and the target) is available, matching could be done by successive searches on each voice in turn. In many music-retrieval or analysis applications, especially where the data has been prepared by encoding a printed score in conventional musical notation, this is possible. But in the general case the data is likely to be imperfectly-specified in terms of its voicing, typically depending on how it is obtained: from audio, for example, even given perfect note-extraction, voicing information is likely to be derivable only approximately, if at all. Therefore, we need to allow for temporal gaps between musical events in the matched pattern.

When we consider the approximate version of this problem we do not require a perfect matching but a matching that is good enough to satisfy certain criteria. The problem of finding substrings of a text similar to a given pattern has been extensively studied in recent years because it has a variety of applications including file comparison, spelling correction, information retrieval, searching for similarities among biosequences and computerized music analysis. One of the most common variants of the approximate string matching problem is that of finding substrings that match the pattern with at most *k-differences*. In this case, k defines the approximation extent of the matching (the edit distance with respect to the three edit operations – *mismatch*, *insert*, *delete*). There is another type of approximate matching; δ-approximate matching. It is well known that a musical score can be represented as a string. This can be accomplished by defining the alphabet to be the set of notes in the chromatic or diatonic notation or the set of intervals that appear between notes. These algorithms can be easily used in the analysis of musical works in order to discover similarities between different musical entities that may lead to establishing a "characteristic signature" [CIR98].

In addition, efficient algorithms for computing approximate matching and repetitions of substrings are also used in molecular biology [FLSS93, KMGL88, MJ93] and particularly in DNA sequencing by hybridization, reconstruction of DNA sequences from known DNA fragments, in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species.

Because exact matching may not help us to find occurrences of a particular melody in a musical work due to the transformation of the particular melody throughout the whole musical work we are compelled to use approximate matching that can absorb, to some extent, this transformation and report the occurrences of this melody. The transformation in different occurrences of a particular melody throughout a musical play is translated into errors of different occurrences of a substring with respect to an initial pattern. Quantity δ defines the error margins of such an approximation.

In [CCIMP99], algorithms *Shift-And* and *Shift-Plus* were presented as efficient solutions to find all δ-occurrences of a given pattern in a text. The *Shift-And* algorithm is based on the constant time computation of different states for each symbol in the text by using bitwise techniques. Therefore, the overall complexity is linear to the size of the text. In [IK02], approximate distributed matching problem for polyphonic music is solved in linear time. We must also mention that it is possible to adapt efficient exact pattern matching algorithms to this kind of approximation. For example, in [CILP01] adaptations of the *Tuned-Boyer-Moore* [HS91] and the *Skip-Search* [CLP98] algorithm were presented.

The organization of the paper is as follows. Some definitions are given in section 2. In section 3, δ-occurrence with α-bounded gaps for monophonic music is considered. In section 4 we consider the problem of computing exact matching with α-bounded gaps for polyphonic music. Finally, we give some conclusions and future work in section 5.

2 Definitions

Let Σ be an alphabet. A string is defined as a sequence of zero or more symbols from Σ. The empty string, that is the string with zero symbols, is denoted by ε. The set of all strings over an alphabet Σ is denoted as Σ^*. A string x of length n is represented by the sequence $x_1, x_2, \ldots, x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. We call w a substring of string x if x is of the form uwv for $u, v \in \Sigma^*$. We also say that substring w occurs at position $|u| + 1$ of string x. The starting position of w in x is the position $|u| + 1$ while position $|u| + |w|$ is said to be the end position of w in x. A string w is a prefix of x if x is of the form wu and is a suffix if x is of the form uw.

We define as the concatenation of two strings x and y the string xy. The concatenations of k copies of a string x is denoted by x^k. Note that self-concatenations can result in strings of exponential size. For two strings $x = x_1, x_2, \ldots, x_n$ and $y = y_1, y_2, \ldots, y_m$ such that $x_{n-i+1}, \ldots, x_n = y_1 \ldots y_i$ for some $i \geq 1$, the string $x_1, \ldots, x_n, y_i, \ldots, y_m$ is the superposition of x and y. In this case we say that x and y overlap.

At this point, we are going to give formally the notion of error introduced in approximate string matching. Assume that δ and γ are integers. Two symbols a, b of alphabet Σ are said to be *δ-approximate*, denoted as $a =_\delta b$, if and only if $|a - b| \leq \delta$. We say that two strings x, y are *δ-approximate*, denoted as $x \overset{\delta}{=} y$ if and only if $|x| = |y|$ and $x =_\delta y$.

Two strings x, y are said to be *γ-approximate*, denoted as $x =_\gamma y$, if and only if $|x| = |y|$ and $\sum_{i=1}^{|x|} |x_i - y_i| < \gamma$. Furthermore, we say that two strings x, y are *(δ, γ)-approximate* if both conditions are satisfied.

The error in the first case (δ-approximate) is defined locally for each symbol in a string. In the second case (γ-approximate) the error is defined in a more global sense and allows us to distribute the error on the symbols unevenly.

3 δ-occurrence with α-bounded gaps for monophonic music

The problem of computing δ-occurrence with α-bounded gaps is formally defined as follows: given a string $t = t_1, \ldots, t_n$, a pattern $p = p_1, \ldots, p_m$ and integers α, δ, check whether there is a δ-occurrence of p in t with gaps whose sizes are bounded by constant α (Fig. 1).

The basic idea of the algorithm described in [CIPR00] is the computation of continuously increasing prefixes of pattern p in text t so that finally we compute the δ-occurrence of the whole pattern p. That is, the algorithm is an incremental procedure that is based on dynamic programming. The algorithm is shown in Fig. 2.

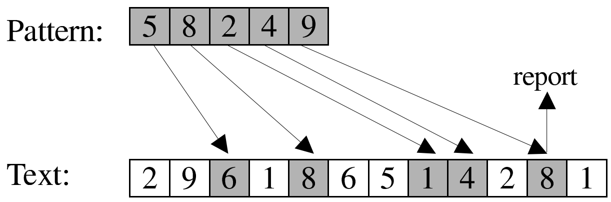


Figure 1: δ-occurrence with α-bounded gaps for $\delta = 1, \alpha = 2$

begin
  $D[0][0] \leftarrow 1$;
  for $i \leftarrow 1$ **to** m **do** $D[i][0] \leftarrow 0$;
  for $j \leftarrow 1$ **to** n **do** $D[0][j] \leftarrow j$;
  for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** n **do**
      if $p[i] =_\delta t[j]$ **and** $j - D[i-1][j-1] \leq \alpha + 1$ **and** $D[i-1][j-1] > 0$
        then $D[i][j] \leftarrow j$;
      elseif $p[i] \neq_\delta t[j]$ **and** $j - D[i][j-1] < \alpha + 1$ **then** $D[i][j] \leftarrow D[i][j-1]$;
      else $D[i][j] \leftarrow 0$;
  for $j \leftarrow 0$ **to** n **do**
    if $D[m][j] > 0$ **then** $\mathrm{OUTPUT}(j)$;
end

Figure 2: Algorithm for δ-occurrence with α-bounded gaps

This algorithm will be adapted to the problem of finding a singular pattern in a singular text (monophonic music) without any major modifications. Fig. 3 shows 2 bars from Michael Niman's piece and a melody which listeners can easily cognize.

If we set the value $\alpha = 3$ (3 gaps allowed), the algorithm can find this melody in the score, while we have to set a large number of k (at least $k = 12$) to find it using k-difference approximate matching algorithms. The time complexity of this algorithm is $O(nm)$, where n is the number of the musical events in the score, which is equivalent to the number of notes in the score since this is monophonic music, and m is the length of the pattern. The running time is shown in Fig. 4.

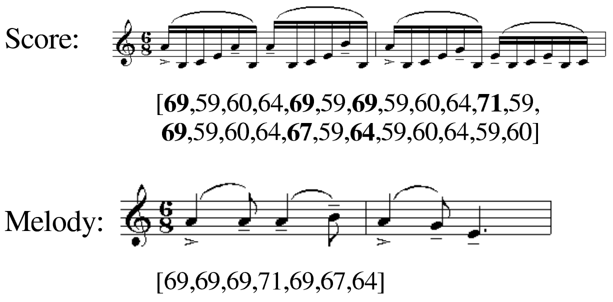


Figure 3: 2 bars from Michael Niman's piece and its melody. If $\alpha \geq 3$, this melody will be found.

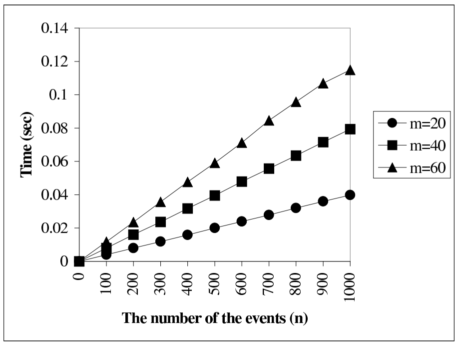


Figure 4: Running time of the algorithm "δ-occurrence with α-bounded gaps for monophonic music". Using a SUN Ultra Enterprise 300MHz running Solaris Unix.

4 Exact matching with α-bounded gaps for polyphonic music

We need to modify the algorithm in order to solve the problem in polyphonic music. Here, we will work on exact matching with α-bounded gaps for polyphonic music, and δ-occurrence will not be considered, as the adjacent pitch does not necessarily mean the most relevant note for a melody. Also, we will suppress the MIDI pitch numbers by dividing by 12 in order to find octave-displaced matches as well. Therefore, 'C' is '1', 'C#' and 'Db' are '2', 'D' is '3', and so on, and the size of alphabet $|\Sigma|$ will be 12.

We are going to use bit arrays and bit-wise operations to deal with several voices at once. Let $Tx[i]$ ($1 \leq i \leq m$, m is the length of a pattern) be a bit array of size $|\Sigma|$ for the position i of the pattern, and $Ty[j]$ ($1 \leq j \leq n$, n is the number of musical events in a plural text) be a bit array of size $|\Sigma|$ for the j-th musical event of the plural text. If $x[i]$ contains a note '8', then the 8th position of $Tx[i]$ will be 0, otherwise 1, where 0 represents 'match' and 1 represents 'mismatch'. Similarly, if $y[j]$ contains notes '3', '4' and '9', then the 3rd and 4th and 9th position of $Ty[j]$ will be 0, otherwise 1. These bit arrays will be used in the searching phase to check whether there is a match or not.

Fig. 5 shows the modified algorithm and the overall time complexity is $O(N+nm)$, where N is the total number of notes in the score, and n is the number of the musical events, and m is the length of the pattern, and Fig. 6 shows its running time. Fig. 7

Preprocessing
begin
  for $j \leftarrow 1$ **to** m **do** $Tx[j] \leftarrow 2^\sigma - 1 - 2^{x[j]}$;
  for $i \leftarrow 1$ **to** n **do**
    $Ty[i] \leftarrow 2^\sigma - 1$;
    for each suppressed pitch p in $y[i]$ **do** $Ty[i] \leftarrow Ty[i]$ & $(2^\sigma - 1 - 2^p)$;
end
Searching
begin
  $D[0][0] \leftarrow 1$;
  for $i \leftarrow 1$ **to** m **do** $D[i][0] \leftarrow 0$;
  for $j \leftarrow 1$ **to** n **do** $D[0][j] \leftarrow j$;
  for $i \leftarrow 1$ **to** m **do**
    for $j \leftarrow 1$ **to** n **do**
      if $(Tx[i] \mid Ty[j]) = Tx[i]$ **and** $j - D[i-1][j-1] \leq \alpha + 1$
        and $D[i-1][j-1] > 0$ **then** $D[i][j] \leftarrow j$;
      elseif $(Tx[i] \mid Ty[j]) \neq Tx[i]$ **and** $j - D[i][j-1] < \alpha + 1$
        then $D[i][j] \leftarrow D[i][j-1]$;
      else $D[i][j] \leftarrow 0$;
  for $j \leftarrow 0$ **to** n **do**
    if $D[m][j] > 0$ **then** OUTPUT(j);
end

Figure 5: Modified algorithm for polyphonic music

and Fig. 9 show examples of the preprocessing phase for 1 bar from Mozart's piano sonata and Debussy's Clair de Lune, respectively, and Fig. 8 and Fig. 10 show their searching phases.

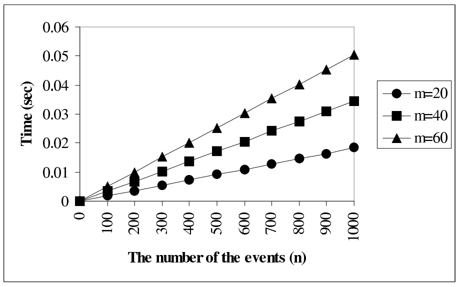
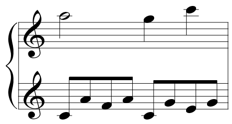


Figure 6: Running time of the modified algorithm for polyphonic music ($N = 4n$). Using a SUN Ultra Enterprise 300MHz running Solaris Unix.

5 Conclusion and further work

Approximate (δ-occurrence) string matching with gaps for monophonic music is solved in $O(nm)$ time, where n is the number of musical events (which is equivalent to the number of notes in a score for monophonic music), and m is the length of a pattern. Exact string matching with gaps for polyphonic music (a plural text and a singular

Score



Event	1	2	3	4	5	6	7	8
Pitch	81 60	69	65	69	79 60	67	84 64	67

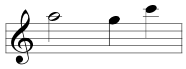
Suppress

Event	1	2	3	4	5	6	7	8
Pitch	10 1	10	6	10	8 1	8	1 5	8

Preprocess

Ty(j)	1	2	3	4	5	6	7	8
1(C)	0	1	1	1	0	1	0	1
2	1	1	1	1	1	1	1	1
3(D)	1	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
5(E)	1	1	1	1	1	1	0	1
6(F)	1	1	0	1	1	1	1	1
7	1	1	1	1	1	1	1	1
8(G)	1	1	1	1	0	0	1	0
9	1	1	1	1	1	1	1	1
10(A)	0	0	1	0	1	1	1	1
11	1	1	1	1	1	1	1	1
12(B)	1	1	1	1	1	1	1	1

Pattern



Pitch	81	79	84

Suppress

Pitch	10	8	1

Preprocess

Tx(i)	1	2	3
1(C)	1	1	0
2	1	1	1
3(D)	1	1	1
4	1	1	1
5(E)	1	1	1
6(F)	1	1	1
7	1	1	1
8(G)	1	0	1
9	1	1	1
10(A)	0	1	1
11	1	1	1
12(B)	1	1	1

Figure 7: Preprocessing phase for 1 bar from a Mozart's piano sonata and a pattern. $(N = 11, n = 8, m = 3)$

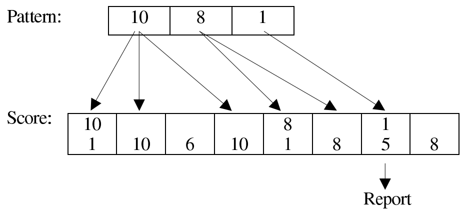
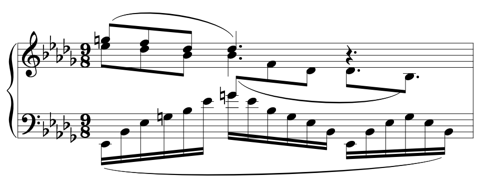


Figure 8: Searching phase using bit-wise operations for 1 bar from the Mozart's piano sonata and the patten. $(\alpha = 3)$

Score



event	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
pitch	79 75 39	46	77 73 51	55	73 70 58	63	73 70 67	63	65 58	55	61 51	46	61 39	46	51	58 55	51	46

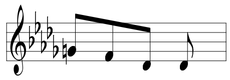
Suppress

event	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
pitch	8 4 4	11	6 2 4	8	2 11 11	4	2 11 8	4	6 11	8	2 4	11	2 4	11	4	11 8	4	11

Preprocess

Ty(j)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1(C)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	0	1	0	1	0	1	1	1	0	1	0	1	1	1	1	1
3(D)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	0	1	0	1	1	0	1	0	1	1	0	1	0	1	0	1	0	1
5(E)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6(F)	1	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8(G)	0	1	1	0	1	1	0	1	1	0	1	1	1	1	1	0	1	1
9	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
10(A)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
11	1	0	1	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0
12(B)	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Pattern



Pitch	67	65	61	61

Suppress

Pitch	8	6	2	2

Preprocess

Tx(i)	1	2	3	4
1(C)	1	1	1	1
2	1	1	0	0
3(D)	1	1	1	1
4	1	1	1	1
5(E)	1	1	1	1
6(F)	1	0	1	1
7	1	1	1	1
8(G)	0	1	1	1
9	1	1	1	1
10(A)	1	1	1	1
11	1	1	1	1
12(B)	1	1	1	1

Figure 9: Preprocessing phase for 1 bar from Clair de Lune and a pattern. ($N = 30, n = 18, m = 4$)

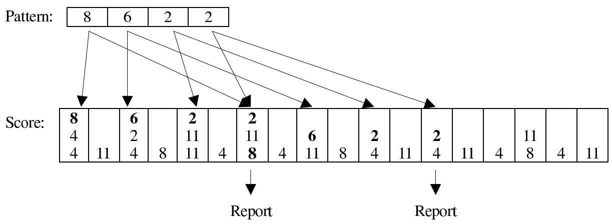


Figure 10: Searching phase using bit-wise operations for 1 bar from Clair de Lune and the patten. ($\alpha = 1$)

pattern) is solved in $O(N + nm)$ time, where N is the total number of notes in a score, and n is the number of musical events ($n \leq N$), and m is the length of a pattern. Using the same technique, exact string matching problem for a plural text and a plural pattern will be solved in $O(N + M + nm)$ time, where M is the total number of notes in the plural pattern. However, we have not solved approximate string matching with gaps for polyphonic music, because "small δ" does not really mean "more relevant" in music. In this particular sense, k-difference algorithms could be more useful, although it is inevitable to have large k and many false matches. We need to design an efficient algorithm for this problem.

References

[CCIMP99] Cambouropoulos, E., Crochemore, M., Iliopoulos, C.S., Mouchard, L., Pinzon, Y.J.: Algorithms for computing approximate repetitions in musical sequences. Proceedings of the 10^{th} Australasian Workshop on Combinatorial Algorithms (AWOCA'99), R. Raman and J. Simpson (editors), Curtin University of Technology, Perth, Western Australia, 129-144.

[CILP01] Crochemore, M., Iliopoulos, C.S., Lecroq, T., Pinzon, Y.J.: Approximate String Matching in Musical Sequences. Proceedings of the Prague Stringology Conference (PSC'01), M. Balik and M. Simanek (editors), Czech Technical University, Collaborative Report DC-2001-06, Prague, Czech Republic, 26-36.

[CIMRTT01] Crochemore, M., Iliopoulos, C.S., Makris, C., Rytter, W., Tsakalidis, A., Tsichlas, K.: Approximate String Matching with Gaps. Nordic Journal of Computing 9, 54-65.

[CIPR00] Crochemore, M., Iliopoulos, C.S., Pinzon, Y.J., Rytter, W.: Finding Motifs with Gaps. Proceedings of the International Symposium on Music Information Retrieval (ISMIR'00), Plymouth, USA, 306-317.

[CIR98] Crawford, T., Iliopoulos, C.S., Raman, R.: String Matching Techniques for Musical Similarity and Melodic Recognition. Computing in Musicology, Vol.11, 73-100.

[CLP98] Charras, C., Lecroq, T., Pehoushek, J.D.: A very fast string matching algorithm for small alphabets and long patterns. Proceedings of the 9^{th} Annual Symposium on Combinatorial Pattern Matching (CPM'98), M. Farach-Colton (editor), number 1448 in Lecture Notes in Computer Science, Piscataway, NJ, Springer-Verlag, Berlin, 55-64.

[FLSS93] Fischetti, V., Landau, G., Schmidt, J., Sellers, P.: Identifying periodic occurrences of a template with applications to protein structure. Information Processing Letters, 45, 11-18.

[HS91] Hume, A., Sunday, D.M.: Fast String Searching. Software-Practice and Experience, 21(11), 1221-1248.

[IK02] Iliopoulos, C.S., Kurokawa, M.: Distributed Matching Problem for Musical Melodic Recognition. Proceedings of the 2002 symposium on AI and Creativity in Arts and Science (AISB'02), A. Cardoso and G. Wiggins (editors), London, 49-56.

[KMGL88] Karlin, S., Morris, M., Ghandour, G., Leung, M.Y.: Efficient Algorithms for molecular sequences analysis. Proc. Natl. Acad. Sci., USA, 85, 841-845.

[LMW02] Lemstrm, K., Meredith, D., Wiggins, G.A.: A geometric approach to computing repeated patterns in polyphonic music. Document submitted to UK Patent office, application number GB 0200203.8.

[MJ93] Milosavljevic, A., Jurka, J.: Discovering simple DNA sequences by the algorithmic significance method. Comput. Appl. Biosci., 9(4), 407-411.

String Regularities with Don't Cares

Costas S. Iliopoulos[1†], Manal Mohamed[1‡], Laurent Mouchard[2],
Katerina G. Perdikuri[3], W. F. Smyth[4] and
Athanasios K. Tsakalidis[3]

[1] Department of Computer Science, King's College London,
London WC2R 2LS, England
{csi,manal}@dcs.kcl.ac.uk
[2] Department of Vegetal Physiology - ABISS, Université de Rouen,
76821 Mont Saint Aignan Cedex, France
Laurent.Mouchard@univ-rouen.fr
[3] Computer Technology Institute, Patras, Greece
perdikur@ceid.upatras.gr, tsak@cti.gr
[4] Algorithms Research Group, Department of Computing & Software,
McMaster University, Hamilton, Ontario, Canada L8S 4K1 and
School of Computing, Curtin University, Perth WA 6845, Australia
smyth@mcmaster.ca

Abstract. We describe algorithms for computing typical regularities in strings $x = x[1..n]$ that contain don't care symbols. For such strings on alphabet Σ, an $O(n \log n \log |\Sigma|)$ worst-case time algorithm for computing the period is known, but the algorithm is impractical due to a large constant of proportionality. We present instead two simple practical algorithms that compute all the periods of every prefix of x; our algorithms require quadratic worst-case time but only linear time in the average case. We then show how our algorithms can be used to compute other string regularities, specifically the covers of both ordinary and circular strings.

Key words: string algorithm, regularities, don't care, period, border, cover.

1 Introduction

Regularities in strings arise in many areas of science: combinatorics, coding and automata theory, molecular biology, formal language theory, system theory, etc. — they thus form the subject of extensive mathematical studies (see e.g. [L83],[P93],[P90]). Perhaps the most conspicuous regularities in strings are those that manifest themselves in the form of repeated subpatterns. A typical regularity, the period u of the string x, grasps the repetitiveness of x, since x is a prefix of a string constructed by

† Partially supported by a Marie Curie fellowship, Wellcome and Royal Society grants.
‡ Supported by EPSRC studentship.

concatenations of u. Here we consider regularity problems that arise from having "don't care" symbols in the string. In particular we study string problems focused on finding the repetitive structures in DNA strings x.

In this paper we also consider a kind of generalized period called a *cover*; that is, a proper substring u of x (if it exists) such that x can be formed by concatenating and overlapping occurrences of u. In the computation of covers, two main problems have been considered in the literature: the *shortest-cover* problem (computing the shortest cover of a given string of length n), and the *all-covers* problem (computing all the covers of a given string). Apostolico, Farach and Iliopoulos [AFI91] introduced the notion of covers and gave a linear-time algorithm for the shortest-cover problem. Breslauer [B92] presented a linear-time on-line algorithm for the same problem. Moore and Smyth [MS95] presented a linear-time algorithm for the all-covers problem. Finally, Li and Smyth [LS02] invented the cover array and described an on-line linear-time algorithm that solves both the shortest-cover and all-covers problems for every prefix of x. In parallel computation, Breslauer [B94] gave an optimal $O(\alpha(n)\log\log n)$-time algorithm for the shortest cover, where $\alpha(n)$ is the inverse Ackermann function; Iliopoulos and Park [IP94] gave an optimal $O(\log\log n)$-time (thus work-time optimal) algorithm for the same problem.

The idea of a cover has been extended. Iliopoulos, Moore and Park [IMP96] introduced the notion of seeds and gave an $O(n\log n)$-time algorithm for computing all the seeds of a given string of length n. For the same problem Ben-Amram, Berkman, Iliopoulos and Park [BBIP94] presented a parallel algorithm that requires $O(\log n)$ time and $O(n\log n)$ work. Apostolico and Ehrenfeucht [AE93] considered yet another problem related to covers.

An interesting extension of string-matching problems with practical applications in the area of DNA sequences results from the introduction of "don't care" symbols. A *don't care* symbol $*$ has the property of matching with any symbol in the given alphabet. For example the string $p = AC*C*$ matches the pattern $q = A*DCT$. Exact string matching with "don't care" symbols was studied by Fischer and Paterson [FP74]. They developed an $O(n\log m\log|\Sigma|)$ time algorithm for finding a pattern of length m in a text of size n over the alphabet $\Sigma\cup\{*\}$. Their method is based on the theoretically fast computation method of convolutions, but it is not efficient in practice. Pinter developed a linear time algorithm for a special case [P85], while Abrahamson generalized Fischer and Paterson's algorithm, using a divide-and-conquer approach that runs in time $O(n\sqrt{m\log m})$ [A87]. See also [LV89].

In this paper we describe two fast, practical algorithms for computing all the periods of every prefix of a given string $x[1..n]$ that contains "don't care" symbols. We prove that the expected running time of these algorithms is linear, though they have quadratic worst-case time complexity for pathological inputs. Then we show how our algorithms can be used to efficiently compute covers of strings with don't cares, both ordinary and circular. The motivation for the above problems comes from many applications to the analysis of DNA sequences that reveal naturally occurring repeated segments within nucleotide sequences. These segments can be concatenated only (periodic) or both concatenated and overlapping (coverable).

2 Background

A *string* is a sequence of zero or more symbols drawn from an alphabet Σ. The set of all nonempty strings over the alphabet Σ is denoted by Σ^+. A string x of length n is represented by $x[1..n] = x[1]x[2]\cdots x[n]$, where $x[i] \in \Sigma$ for $1 \le i \le n$, and $n = |x|$ is the length of x. The empty string is the empty sequence (of zero length) and is denoted by ε; we write $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$ The string xy is a *concatenation* of two strings x and y. The concatenation of k copies of x is denoted by x^k and is called *the k^{th} power of x.*

A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$, a *proper prefix* if $u \in \Sigma^+$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$. A string u that is both a proper prefix and a suffix of x is called *a border* of x.

If x has a nonempty border, it is called *periodic*. Otherwise, x is is said to be *primitive*. The empty string is a trivial border of x. Let u denote a border of x of length ℓ where $1 \le \ell \le n-1$; then $p = n - \ell$ is called *a period* of x. Clearly, p is a period of x if $x_i = x_{i+p}$ whenever $1 \le i, i+p \le n$. Another equivalent definition may be given as: p is a period of x if and only if $x[1..p] = x[n-p+1..n]$. The latter definition shows that each word x has a minimum period called *the period* of x. For example, the string $x = ababab$ has two borders $u_1 = ab$ and $u_2 = abab$; thus x has two periods 4 and 2, where 2 is *the* period of x.

A substring u is said to be a *cover* of a given string x if every position of x lies within an occurrence of a string u within x. If, in addition, $|u| < |x|$, we call u a *proper cover* of x. For example, x is always a cover of x. and $u = aba$ is a proper cover of $x = abaababa$.

An array $\beta[1..n]$ is called the *border array* of $x[1..n]$, where for $i = 1, 2, \ldots, n$, $\beta[i]$ gives the length of the longest border of $x[1..i]$. Furthermore, since every border of a border of x is itself a border of x, β actually describes all the borders of every prefix of x. The border array can be computed in linear time using the classical failure function algorithm [AHU74].

Recently Li and Smyth [LS02] discovered the cover array $\gamma[1..n]$, where $\gamma[i]$ gives the length of the longest cover of $x[1..i]$. The cover array similarly encapsulates all the covers of every prefix of x and can also be computed in linear time.

This paper deals with strings that can contain occurrences of the *don't care* symbol, denoted by "$*$". This symbol matches any other symbol of the alphabet. Two symbols a and b match ($a \approx b$) if they are equal, or if one of them is a don't care symbol. Notice that the relation $\approx$ is not transitive ($a \approx *, * \approx b \not\Rightarrow a \approx b$).

3 Computing the Failure Function

A theoretical $O(n \log n \log |\Sigma|)$ time algorithm for computing the period of a given string x that contains don't care symbols can be achieved by using a "convolution" procedure [FP74] between two strings x and X. Assuming that x is the given string (of length n), we create a string X by adding n don't care symbols, thus doubling the length of x. We compute the convolution of x and X by shifting x to the right by one character. The product u of the convolution is the period of the string x (for further information see [FP74]). This algorithm is impractical as it has a very large constant hidden in its asymptotic time complexity.

In this section we present two fast and practical algorithms for computing the border array $\beta[1 \dots n]$ of a given string x that contains don't care symbols.

As noted earlier, the standard failure function method, based on the fact that "a border of a border of a string x is necessarily a border of x", cannot be used to calculate the border array of a string containing don't care symbols. This follows from the nontransitivity of the $\approx$ relation. For example, if $x = a**ca$, then we have

$$u_l = a** \approx u_r = *ca,$$

where u_l and u_r are respectively the left and right borders of x of length 3; note that $v_l = a* \approx **$ is a border of u_l, but $a* \neq ca$, which means that v_l is not a border of u_r, hence not of x.

Despite the fact that we cannot make use of the standard failure function method, it is quite easy to notice that there is no nonempty border b of $x[1..i+1]$ that is not equal to some $b'x[i+1]$, where b' is a border of $x[1 \dots i]$. Moreover, let the borders of $x[1..i]$ be

$$\beta^1[i], \beta^2[i] \dots \beta^k[i]$$

where $\beta^1[i]$ is the the length of the border of $x[1 \dots i]$ (the longest border) and $\beta^k[i] = 0$ is the length of the empty border .Then each border of $x[1 \dots i+1]$ is equal to either $\beta^j[i] + 1$ for some $1 \leq j \leq k$ or 0.

The above states the rule used by algorithm FAILURE-FUNCTION-1() to calculate the value of the border array of a given string x that contains don't care symbols.

FAILURE-FUNCTION-1(x)
$S \leftarrow \emptyset$ S is a singly-linked list of nonzero border lengths
$\beta[1] \leftarrow 0$
For $i \leftarrow 1$ **To** $n-1$ **Do**
  For each $b \in S$ **Do**
    If $x[i+1] \approx x[b+1]$ **Then**
      replace_current($S, b+1$)
    Else delete_current(S)
  If $x[i] \approx x[1]$ **Then** add_after_current(S,1)
  If $S \neq \emptyset$ **Then** $\beta[i+1] \leftarrow$top(S)
  Else $\beta[i+1] \leftarrow 0$
END FAILURE-FUNCTION-1

Figure 1: FAILURE-FUNCTION-1 algorithm.

The algorithm maintains a list S of all possible nonzero border lengths. At the beginning of iteration i, S contains all possible nonzero border lengths of $x[\dots i]$. The algorithm tries to extend each possible border b in S by comparing the value of $x[i+1]$ and the value of $x[b+1]$. If the two values are equal or one of them is a don't care symbol, the value b in S is replaced by $b+1$. Otherwise, b is deleted from the list. If $x[i+1]$ is equal to $x[1]$ or $*$, a border of length 1 has to be added to S. Finally, each iteration i terminates by assigning the value at the top of the list S that is the length of the longest border of $x[1\ldots i+1]$ to $\beta[i+1]$. If the list S is empty, then the length of the longest border is 0 ($\beta[i+1]=0$). Note that at this stage , S contains the lengths of all possible nonzero borders of $x[1\ldots i+1]$ in descending order.

Each position i such that $x[i]=x[1]$ or $*$ is a candidate to start a new border. Hence Algorithm FAILURE-FUNCTION-2() tries to speed up the computation of the failure function by a simple linear preprocessing of the input string x. For each position i we count the previous occurrences of $x[1]$'s and $*$'s. And we introduce a pointer that points to the previous occurrence. The algorithm then modifies the standard failure function method to calculate the border array β. FAILURE-FUNCTION-2 starts by setting the value of $\beta[0]$ to -1, a convention which is compatible with the algorithm. Then $n-1$ iterations follow. In each iteration i, the algorithm tries to extend the current border b by comparing the value of $x[i+1]$ and the value of $x[b+1]$ where b is the length of the border of $x[1\ldots i]$. If the two values are equal or one of them is a don't care symbol, the value of $\beta[i]$ is set to $b+1$. Otherwise, the algorithm tries to follow the basic failure function method by trying to extend the border of the current border. More work needs to be done in each attempt to ensure the right answer:

- The algorithm has to eliminate the possibility of having a border whose length is greater than that of the border of the border. That is, having

  $$x[1\ldots i-j+2]\approx x[j\ldots i+1]$$

  for some j such that $\beta[b]<i-j+1<b$. The algorithm uses the preprocessed informations to find each position j such that $x[j]=x[1]$ or $*$. Clearly, the number and the positions of the j's can be calculated in constant time. The algorithm examines each j in ascending order to find the first j that satisfies the above condition. If such a j exists, then the iteration ends by assigning $i-j+2$ to $\beta[i+1]$.

- Recall that the nontransitivity of the $\approx$ relation means that the statement "the border of the border is a border" may not be true. Observe that nontransitivity can occur only if a don't care symbol was part of the comparison. Then only in such cases does the algorithm need to recheck the positions that could cause a nontransitivity. That is, if $x[i+1]\approx x[\beta[b]]$, then the algorithm still needs to check all the solid characters in the right border; that have been compared with the don't care symbol during the calculation; against the coresponding characters in the left border. These positions are marked during the calculations and stored in a special stack S. Positions are popped from and pushed onto S depending on the length of the current border.

For example, let $x=a**cabcdabc*abca$ and the value of the border array be as follows:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$x[i]$	a	$*$	$*$	c	a	b	c	d	a	b	c	$*$	a	b	c	a
$\beta[i]$	0	1	2	3	3	2	3	0	1	2	3	4	5	6	7	5

At step 7 ($i=7$) we had failed to extend the current border after comparing $x[4]=$ 'c' with $x[8]=$ 'd'. At the same time we could not find any j that satisfies the first

FAILURE-FUNCTION-2(x)
$\beta[0] \leftarrow -1$
For $i \leftarrow 0$ **To** $n-1$ **Do**
  $b \leftarrow \beta[i]$
  If $x(i+1) \approx x(b+1)$ **Then** $\beta[i+1] \leftarrow b+1$
  Else
    While $b \geq 0$ **And** $[x(i+1) \neq x(b+1)$ **Or** check_stack_fail()$]$ **Do**
      For each j such that $\beta[b] < i-j+1 < b$ **And** $x[j] \approx x[1]$ **Do**
        If $x[j..i+1] \approx x[1..i-j+2]$ **Then**
          $b \leftarrow i-j+2$
          Quit The While Loop
      $b \leftarrow \beta[b]$
    $\beta[i+1] \leftarrow b$
END FAILURE-FUNCTION-2

Figure 2: FAILURE-FUNCTION-2 algorithm.

condition. So we tried to extend the border of the border which equals 3 ($\beta[7] = 3$). Since $x[8] \neq x[4]$, we tried to extend the border of the border of the border which equals 2 ($\beta[3] = 2$). Although $x[8] \approx x[3]$, we still need to check according to the algorithm the value at position 1 with the corresponding value at position 6. Since they are not equal, the value of $\beta[8]$ can not be 3 and so we have to carry on . Note that the value 1 had been inserted into the stack after comparing the '$*$' at position 2 with the 'a' at position 1 at step 1.

At step 15, where $x[16] \neq x[8]$, we had failed again to extend the current border. According to the algorithm we have to eliminate the possibility of having a longer border than the border of the border; that is, finding j that satisfies the first condition. In our example, we found $j = 12$. Note that

$$\beta[b] = 3 < i - j + 1 = 15 - 12 + 1 = 4 < b = 7$$

and $x[12] = *$. After finding j we need to compare $x[12\ldots16]$ with $x[1\ldots5]$. Since they are equal the value of $\beta[16]$ becomes 5.

4 Expected Running Time Analysis

Here we will show that the expected number of borders of a string is bounded by a constant. We suppose that the alphabet Σ consists of ordinary letters $1\ldots\sigma-1$ together with the don't care symbol $*$. First we consider the probability of two symbols of a string being equal. Equality occurs in the following cases:

Symbol	Equal to	Number of cases
$*$	$\sigma \in \{1,\ldots,\alpha-1\}$	$\alpha - 1$
$\sigma \in \{*,1,\ldots,\alpha-1\}$	$*$	α
$\sigma \in \{1,\ldots,\alpha-1\}$	$\sigma \in \{1,\ldots,\alpha-1\}$	$\alpha - 1$

Thus the total number of equality cases is $3\alpha - 2$ and the number of overall cases is α^2. Therefore the probability of two symbols of a string being equal is

$$\frac{3\alpha - 2}{\alpha^2}$$

Now let consider the probability of string x having a border of length k. One can see

$$P[x_1 \ldots x_k = x_{n-k-1} \ldots x_n] = P[x_1 = x_{n-k-1}] \ldots P[x_k = x_n] = \left(\frac{3\alpha - 2}{\alpha^2}\right)^k$$

From this it follows that the expected number of borders is

$$\Sigma_{k=1}^{n-1} \left(\frac{3\alpha - 2}{\alpha^2}\right)^k < 3.5$$

The algorithm, at iteration i, performs k_i steps, where k_i is the number of the borders of $x[1..i]$. Thus the overall expected time complexity is

$$\Sigma_{k=1}^{n-1} k_i.$$

Since the expected value of each k_i is bounded by 3.5, therefore the expected time of the two border algorithms is O(n).

5 Experimental Results

Using random strings over various alphabet sizes (with the $*$ symbol treated as an additional random letter), we ran FAILURE-FUNCTION-1() and FAILURE-FUNCTION-2(). The running time was calculated for each execution. We used a SUN Ultra Enterprise 300MHz running Solaris Unix. The reported times are the calculation time in seconds, measured by calling the a clock() routine (Figures 3 and 4).

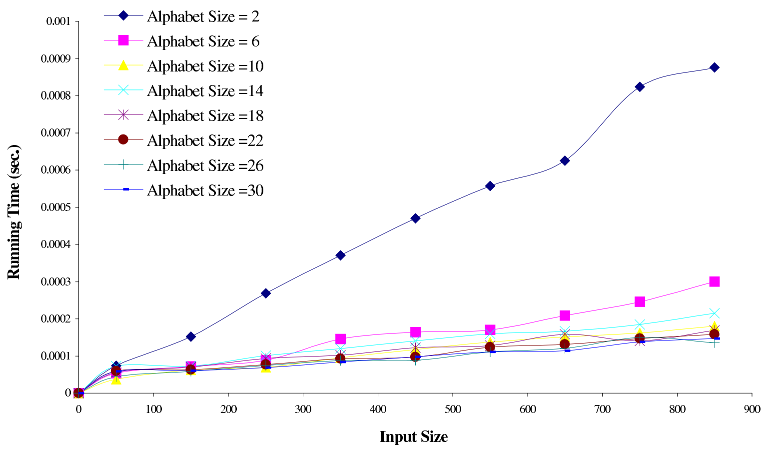


Figure 3: Timing curves for the FAILURE-FUNCTION-1 Procedure.

In general, it seems that the heuristic employed in FAILURE-FUNCTION-2 is effective for random strings on small alphabets (therefore containing a high proportion

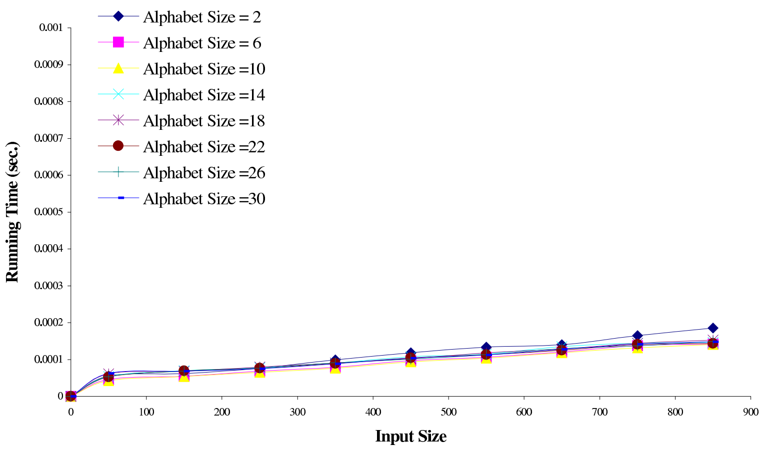


Figure 4: Timing curves for the FAILURE-FUNCTION-2 Procedure.

FIND-COVERS(x)
Compute borders $B = \{b_1, \ldots, b_k\}$ of x in ascending order of length
For each adjacent pair of borders, b_i and b_{i+1}, **Do**
    If b_i covers b_{i+1} **Then** check whether it covers x
    Else $i \leftarrow i + 1$
END FIND-COVERS

Figure 5: FIND-COVERS algorithm.

of don't care symbols), but makes little difference for larger alphabets that have a correspondingly low proportion of don't cares.

Note that our experiments confirm Section 4's theoretical result that the expected case bahaviour of the algorithms is linear in string length.

6 Computing the Covers

In this section we present an algorithm for computing all the covers of a given string x, bearing in mind that we allow possible overlaps. This means that in the example $p = AC{*}ACA{*}AA{*}ACA$, the pattern $q = ACA$ is an overlapping cover of the string p. The algorithm we present consists of 2 stages. The first stage is a preprocessing phase where we compute the borders of the given string x. Suppose we find the following nonempty borders $b_1, b_2, \ldots, b_k$, listed in ascending order.

In the second stage we perform the following check: *for two borders b_i and b_{i+1}, if b_i covers b_{i+1} we check whether b_i also covers string x. If not we continue this process for the rest of the adjacent pairs of borders.*

In order to precompute the borders we use Algorithm *ALL-BORDERS()*. Using the previously computed borders, the procedure that finds the covers of a given string x is as follows:

Theorem 6.1 *Given a string x that contains don't care symbols, we can find a longest cover u of x in linear expected time.*

7 Computing the Covers of Circular DNA Strings

In some computational biology applications (for example, DNA sequencing by hybridization), it is convenient to regard the DNA sequence as a circular string (Fig. 6). Given a circular DNA string and a window that limits the region of DNA that we are able to study, the computation of covers in the sequence becomes a difficult task. In that case the computation of seeds (see [BBIP94]) does not work and we need a new approach.

Bearing in mind the scheme of a circular DNA string and the algorithms for the computation of the failure function that we have already described, it is easy to see that the computation of the covers in a circular DNA sequence can be easily solved using the failure function technique. More precisely the problem of the computation of covers can be solved if we compute the failure function two times, once forward and once backward.

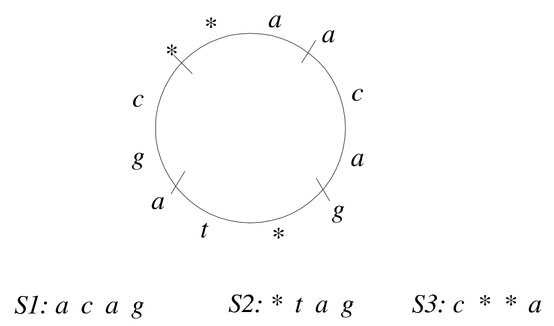


Figure 6: A circular string x and three substrings S1, S2, S3, as seen from a window of four characters length.

Conclusions

We have presented two linear expected-time algorithms for computing all the borders (hence all the periods) of a given string containing don't care symbols. We have then shown how to apply the border calculation to compute the covers of ordinary and circular strings, also containing don't care symbols.

An open problem is the calculation of every border of every prefix of x in $O(n \log n)$ worst-case time.

References

[A87] Abrahamson, K.: Generalized string matching, SIAM J.Computing, 16, 1039-1051.

[AE93] Apostolico, A., Ehrenfeucht, A.: Efficient detection of quasiperiodicities in strings, Theo. Comp. Sci, 119, 247-265.

[AFI91] Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings, Inform. Process. Lett., 39, 17-20.

[AHU74] Aho, Alfred V., Hopcroft, John E., Ullman, Jeffey D.: The Design & Analysis of Computer Algorithms, Addison-Wesley.

[B92] Breslauer, D.: An on-line string superprimitivity test, Inform. Process. Lett., 44, 345-347.

[B94] Breslauer, D.: Testing string superprimitivity in parallel, Inform. Process. Lett., 49, 235-241.

[BBIP94] Ben-Amram, A.M., Berkman, O.C.S., Iliopoulos, C.S., Park, K.: The subtree max gap problem with application to parallel string covering, Proc. 5th ACM-SIAM Symp. Discrete Algorithms, 501-510.

[FP74] Fischer, M., Paterson, M. : String matching and other products, Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings, 7, 113-125.

[IMP96] Iliopoulos, C.S., Moore, D.W.G., Park, K.: Covering a string, Algorithmica, 16, 288-297.

[IP94] Iliopoulos, C.S., Park, K.: An optimal $O(\log\log n)$ time algorithm for parallel superprimitivity testing, Journal of the Korean Information Science Society, 21-8, 1400-1404.

[L83] Lothaire, M. : Combinatorics on Words, Addison-Wesley, Reading, Mass.

[LS02] Yin Li, Smyth, W.F.: Computing the cover array in linear time, Algorithmica, 32-1, 95-106.

[LV89] Landau, G.M., Viskin, U.: Fast parallel and serial approximate string matching, Journal of Algorithms, 10, 157-169.

[MS95] Moore, D.W.G., Smyth, W.F.: A correction to "Computing the covers of a string in linear time", Inform. Process. Lett., 54,101-103.

[P85] Pinter, R.: Efficient string matching with don't-care patterns, Combinatorial Algorithms on Words, NATO ASI Series, F12, Springer-Verlag, 11-29.

[P90] Pevzner, P.A.: Statistical analysis of genetics texts, Computer Analysis of Genetics Texts, Chapter 2, Ed. M.D. Frank-Kamenetzkii, Nauka, Moscow, 36-80 (in Russian).

[P93] Pevzner, P.A.: Overlapping word paradox and Conway Equation, Supercomputing, Bioinformatics, and Complex Genome Analysis, World Scientific, Ed. C. Cantor, J. Fickett, R. Robbins, and H. Lim, 71-78.

Bidirectional Construction of Suffix Trees

Shunsuke Inenaga

Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan

e-mail: `s-ine@i.kyushu-u.ac.jp`

Abstract. String matching is critical in information retrieval since in many cases information is stored and manipulated as strings. Constructing and utilizing suitable data structures for text strings, we can solve the string matching problem efficiently. Such structures are called *index structures*. The *suffix tree* is certainly the most widely-known and extensively-studied structure of this kind. In this paper, we present a linear-time algorithm for bidirectional construction of suffix trees.

1 Introduction

Pattern matching on strings is of central importance to Theoretical Computer Science. The pattern matching problem is to examine whether a given pattern string p matches a text string w. This problem can be solved in $O(|p|)$ time, by using a suitable *index structure*.

The most basic index structure seems to be the suffix trie, by whose nodes all substrings of a given string w are recognized. Probably the structure is the easiest to understand, but its only, however biggest drawback is that its space requirement is $O(|w|^2)$.

This fact led the introduction of more space-economical ($O(|w|)$-spaced) structures such as the suffix tree [23, 19, 22, 12], the directed acyclic word graph (DAWG) [3, 7, 2], the compact directed acyclic word graph (CDAWG) [4, 9, 15, 13, 16], the suffix array [18], and some other variants. Among those, suffix trees are possibly most widely-known and extensively-studied [8, 12], perhaps because there are a 'myriad' [1] of applications for them.

Construction of suffix trees has been considered in various contexts: Weiner [23] invented the first algorithm that constructs suffix trees in linear time; McCreight [19] proposed a more space-economical algorithm than Weiner's; Chen and Seiferas [6] showed an efficient modification of Weiner's algorithm; Ukkonen [22] introduced an on-line algorithm to construct suffix trees, which Giegerich and Kurtz [11] regarded as "the most elegant"; Farach [10] considered optimal construction of suffix trees with large alphabets; Breslauer [5] gave a linear-time algorithm for building the suffix tree of a given trie that stores a set of strings; Inenaga et al. [14] presented an on-line algorithm that simultaneously constructs both the suffix tree of a string and the DAWG of the reversed string.

In this paper we explore *bidirectional construction* of suffix trees. Namely, the algorithm we propose allows us to update the suffix tree of a string w to the suffix tree of a string xwy, where x, y are any strings. We also show that our algorithm runs in linear time and space with respect to the length of a given string.

Some related work can be seen in literature: Stoye [20, 21] invented variant of suffix trees, called *affix trees*. He proposed an algorithm for bidirectional construction of affix trees, and Maaß [17] improved the time complexity of the algorithm to $O(|w|)$.

2 Suffix Trees

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of string $w = xyz$, respectively. The sets of prefixes, factors, and suffixes of a string w are denoted by $\mathit{Prefix}(w)$, $\mathit{Factor}(w)$, and $\mathit{Suffix}(w)$, respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$. Let $S \subseteq \Sigma^*$. The cardinality of S is denoted by $|S|$. For any string $u \in \Sigma^*$, $Su^{-1} = \{x \mid xu \in S\}$.

Let $w \in \Sigma^*$. We define an equivalence relation $\equiv^L_w$ on Σ^* by

$$x \equiv^L_w y \Leftrightarrow \mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)y^{-1}.$$

The equivalence class of a string $x \in \Sigma^*$ with respect to $\equiv^L_w$ is denoted by $[x]^L_w$. Note that all strings not belonging to $\mathit{Factor}(w)$ form one equivalence class under $\equiv^L_w$. This equivalence class is called the *degenerate* class. All other classes are said to be *non-degenerate*.

Proposition 1 ([14]) *Let $w \in \Sigma^*$ and $x, y \in \mathit{Factor}(w)$. If $x \equiv^L_w y$, then either x is a prefix of y, or vice versa.*

Proof. By the definition of $\equiv^L_w$, we have $\mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)y^{-1}$. There are three cases to consider:

(1) When $|x| = |y|$. Obviously, $x = y$ in this case. Thus $x \in \mathit{Prefix}(y)$ and $y \in \mathit{Prefix}(x)$.

(2) When $|x| > |y|$. Let u be an arbitrary string in $\mathit{Prefix}(w)$. Assume $u = sx$ with $s \in \Sigma^*$. Then $s \in \mathit{Prefix}(w)x^{-1}$, which results in $s \in \mathit{Prefix}(w)y^{-1}$. Hence, there must exist a string $v \in \mathit{Prefix}(w)$ such that $v = sy$. By the assumption that $|x| > |y|$, we have $|u| > |v|$. From the fact that both u and v are in $\mathit{Prefix}(w)$, it is derived that $v \in \mathit{Prefix}(u)$. Consequently, $y \in \mathit{Prefix}(x)$.

(3) When $|x| < |y|$. By a similar argument to the one in Case (2), we have $x \in \mathit{Prefix}(y)$.

$\square$

For any string $x \in \mathit{Factor}(w)$, the longest member in $[x]^L_w$ is denoted by $\overset{w}{\overrightarrow{x}}$.

Proposition 2 ([14]) *Let $w \in \Sigma^*$. For any $x \in \mathit{Factor}(w)$, there uniquely exists a string $\alpha \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\alpha$.*

Proof. Let $\overset{w}{\overrightarrow{x}} = x\alpha$ with $\alpha \in \Sigma^*$. For the contrary, assume there exists a string $\beta \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\beta$ and $\beta \neq \alpha$. By Proposition 1, either $x\alpha \in \mathit{Prefix}(x\beta)$ or $x\beta \in \mathit{Prefix}(x\alpha)$ must stand, since $x\alpha \equiv^L_w x\beta$. However, neither of them actually holds since $|\alpha| = |\beta|$ and $\alpha \neq \beta$, which yields a contradiction. Hence, α is the only string satisfying $\overset{w}{\overrightarrow{x}} = x\alpha$. □

Proposition 3 *Let $w \in \Sigma^*$ and $x \in \mathit{Factor}(w)$. Assume $\overset{w}{\overrightarrow{x}} = x$. Then, for any $y \in \mathit{Suffix}(x)$, $\overset{w}{\overrightarrow{y}} = y$.*

Proof. Assume contrarily that there uniquely exists a string $\alpha \in \Sigma^+$ such that $\overset{w}{\overrightarrow{y}} = y\alpha$. Since $y \in \mathit{Suffix}(x)$, x is always followed by α in w. It implies that $\mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)(x\alpha)^{-1}$, and therefore we have $x \equiv^L_w x\alpha$. That $|\alpha| > 0$ means that $\overset{w}{\overrightarrow{x}}$ is not the longest in $[x]^L_w$; a contradiction. Hence, $\overset{w}{\overrightarrow{y}} = y$. □

Proposition 4 *Let $w \in \Sigma^*$. For any string $x \in \mathit{Suffix}(w)$, $\overset{w}{\overrightarrow{x}} = x$.*

Proof. Let $y \in \Sigma^*$ be an arbitrary string such that $x \equiv^L_w y$ and $x \neq y$. Then, we have $\mathit{Prefix}(w)x^{-1} = \mathit{Prefix}(w)y^{-1}$. Because $x \in \mathit{Suffix}(w)$, $y \in \mathit{Prefix}(x) - \{x\}$ and thus $|x| > |y|$. Hence, $\overset{w}{\overrightarrow{x}} = x$. □

The number of strings in $\mathit{Factor}(w)$ is $O(|w|^2)$. For example, consider string $a^n b^n$. However, for any string $w \in \Sigma^*$, the number of strings x such that $x = \overset{w}{\overrightarrow{x}}$ is $O(|w|)$. The following lemma gives a tighter upperbound.

Lemma 1 ([3, 4]) *Assume that $|w| > 1$. The number of the non-degenerate equivalence classes in $\equiv^L_w$ is at most $2|w| - 1$.*

In the following, we define the suffix tree of a string $w \in \Sigma^*$, denoted by $\mathit{STree}(w)$, on the basis of the above-mentioned equivalence classes. We define it as an edge-labeled tree (V, E) with $E \subseteq V \times \Sigma^+ \times V$ where the second component of each edge represents its label. We also give a definition of the *suffix links*, kinds of failure functions, frequently utilized for time-efficient construction of suffix trees [23, 19, 22].

Definition 1 *$\mathit{STree}(w)$ is the tree (V, E) such that*

$$\begin{aligned} V &= \{\overset{w}{\overrightarrow{x}} \mid x \in \mathit{Factor}(w)\}, \\ E &= \{(\overset{w}{\overrightarrow{x}}, a\beta, \overset{w}{\overrightarrow{xa}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overrightarrow{xa}} = xa\beta,\ \mathit{and}\ \overset{w}{\overrightarrow{x}} \neq \overset{w}{\overrightarrow{xa}}\}, \end{aligned}$$

and its suffix links are the set

$$F = \{(\overset{w}{\overrightarrow{ax}}, \overset{w}{\overrightarrow{x}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \mathit{and}\ \overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overrightarrow{x}}\}.$$

The node $\overset{w}{\overrightarrow{\varepsilon}} = \varepsilon$ is called the *root* node of $\mathit{STree}(w)$. When a node $\overset{w}{\overrightarrow{x}}$ is of out-degree zero, it is said to be a *leaf* node. Each leaf node corresponds to a string in $\mathit{Suffix}(w)$. If $x \in \mathit{Factor}(w)$ satisfies $x = \overset{w}{\overrightarrow{x}}$, x is said to be represented on *explicit* node $\overset{w}{\overrightarrow{x}}$. If $x \neq \overset{w}{\overrightarrow{x}}$, x is said to be on an *implicit* node. $\mathit{STree}(\texttt{coco})$ and $\mathit{STree}(\texttt{cocoa})$ are displayed in Figure 1.

It derives from Lemma 1 that:

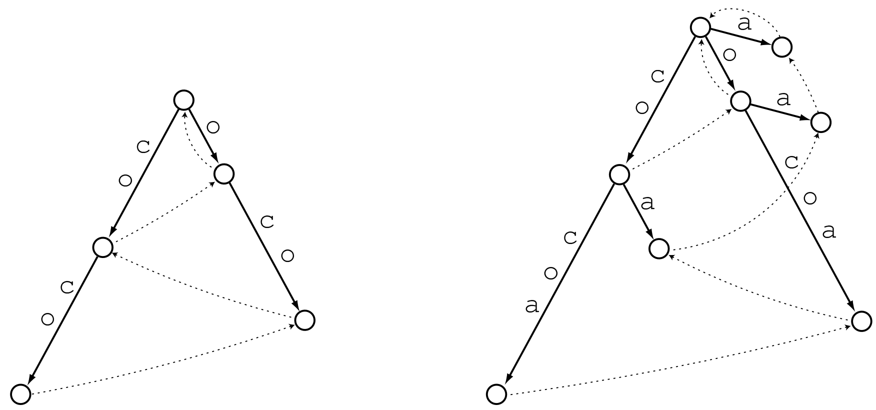


Figure 1: $STree(\texttt{coco})$ on the left, and $STree(\texttt{cocoa})$ on the right. Solid arrows represent edges, while dotted arrows denote suffix links.

Theorem 1 ([19]) *Let $w \in \Sigma^*$. Let $STree(w) = (V, E)$. Assume $|w| > 1$. Then $|V| \leq 2|w| - 1$ and $|E| \leq 2|w| - 2$.*

Weiner's algorithm [23] and McCreight's algorithm [19] construct the suffix tree defined above, $STree(w)$. On the other hand, Ukkonen's algorithm constructs a slightly different version, which is suitable for his algorithm.

As a preliminary to define the modified suffix tree, we firstly introduce a relation X_w over Σ^* such that

$$X_w = \{(x, xa) \mid x \in Factor(w) \text{ and } a \in \Sigma \text{ is unique such that } xa \in Factor(w)\}.$$

Let $\equiv'^L_w$ be the equivalence closure of X_w, i.e., the smallest superset of X_w that is symmetric, reflexive, and transitive.

Proposition 5 ([14]) *For any string $w \in \Sigma^*$, $\equiv^L_w$ is a refinement of $\equiv'^L_w$.*

Proof. Let x, y be any strings in $Factor(w)$ and assume $x \equiv^L_w y$. According to Proposition 1, we firstly assume that $x \in Prefix(y)$. It follows from Proposition 2 that there uniquely exist strings $\alpha, \beta \in \Sigma^*$ such that $\overset{w}{\overrightarrow{x}} = x\alpha$ and $\overset{w}{\overrightarrow{y}} = y\beta$. Note that $\beta \in Suffix(\alpha)$. Let $\gamma \in \Sigma^*$ be the string satisfying $\alpha = \gamma\beta$. Then γ is the sole string such that $x\gamma = y$. By the definition of $\equiv'^L_w$, we have $x \equiv'^L_w y$. A similar argument holds in case that $y \in Prefix(x)$. □

Corollary 1 ([14]) *For any string $w \in \Sigma^*$, every equivalence class under $\equiv'^L_w$ is a union of one or more equivalence classes under $\equiv^L_w$.*

For a string $x \in Factor(w)$, the longest string in the equivalence class with respect to x under $\equiv'^L_w$ is denoted by $\overset{w}{\Longrightarrow}\!\!\!\!\!x$.

The next proposition corresponds to Proposition 3

Proposition 6 *Let $w \in \Sigma^*$ and $x \in Factor(w) - Suffix(w)$. Assume $\overset{w}{\underset{x}{\Longrightarrow}} = x$. Then, for any $y \in Suffix(x)$, $\overset{w}{\underset{y}{\Longrightarrow}} = y$.*

Proof. Since $\overset{w}{\overset{\Longrightarrow}{x}} = x$ and $x \notin \mathit{Suffix}(w)$, there are at least two characters $a, b \in \Sigma$ such that $xa, xb \in \mathit{Factor}(w)$ and $a \neq b$. Since $y \in \mathit{Suffix}(x)$, y is also followed by both a and b in the string w. Thus $\overset{w}{\overset{\Longrightarrow}{y}} = y$. □

Remark that the precondition of the above proposition slightly differs from that of Proposition 3. Namely, when x is a suffix of w, this proposition does not always hold.

From here on, we explore some relationship between $\overset{w}{\overrightarrow{(\cdot)}}$ and $\overset{w}{\overset{\Longrightarrow}{(\cdot)}}$.

Lemma 2 ([14]) *Let $w \in \Sigma^*$. For any string $x \in \mathit{Factor}(w)$, $\overset{w}{\overrightarrow{x}}$ is a prefix of $\overset{w}{\overset{\Longrightarrow}{x}}$. If $\overset{w}{\overrightarrow{x}} \neq \overset{w}{\overset{\Longrightarrow}{x}}$, then $\overset{w}{\overrightarrow{x}} \in \mathit{Suffix}(w)$.*

Proof. We can prove that $\overset{w}{\overrightarrow{x}} \in \mathit{Prefix}(\overset{w}{\overset{\Longrightarrow}{x}})$ by Proposition 1 and Corollary 1. Now suppose $\overset{w}{\overrightarrow{x}} \neq \overset{w}{\overset{\Longrightarrow}{x}}$. Let $\overset{w}{\overrightarrow{x}} = x\beta$ with $\beta \in \Sigma^+$. Supposing $\overset{w}{\overset{\Longrightarrow}{x}} = x\alpha$ with $\alpha \in \Sigma^+$, we have $\beta \in \mathit{Prefix}(\alpha)$. Let $\beta\gamma = \alpha$ with $\gamma \in \Sigma^*$. By the assumption $\overset{w}{\overrightarrow{x}} \neq \overset{w}{\overset{\Longrightarrow}{x}}$, we have $x\beta \not\equiv^L_w x\alpha$, although γ is the sole string that follows $x\beta$ in w since $\overset{w}{\overset{\Longrightarrow}{x}} = x\alpha$. Therefore, x must be a suffix of w, which is followed by *no* character. □

For example, consider string $w = \texttt{coco}$. Then, $\overset{w}{\overrightarrow{\texttt{co}}} = \texttt{co}$ but $\overset{w}{\overset{\Longrightarrow}{\texttt{co}}} = \texttt{coco}$, where $\texttt{co}$ is a suffix of $\texttt{coco}$.

Lemma 3 *Let $w \in \Sigma^*$ and $x \in \mathit{Suffix}(w)$. If $x \notin \mathit{Prefix}(y)$ for any string $y \in \mathit{Factor}(w) - \{x\}$, then $\overset{w}{\overrightarrow{x}} = \overset{w}{\overset{\Longrightarrow}{x}}$.*

Proof. The precondition implies that there is no character $a \in \Sigma$ satisfying $xa \in \mathit{Factor}(w)$. Thus we have $\overset{w}{\overset{\Longrightarrow}{x}} = x$. On the other hand, we obtain $\overset{w}{\overrightarrow{x}} = x$ by Proposition 4, because $x \in \mathit{Suffix}(w)$. Hence $\overset{w}{\overrightarrow{x}} = \overset{w}{\overset{\Longrightarrow}{x}}$. □

Lemma 4 *Let $w \in \Sigma^*$ with $|w| = n$. Assume that the last character $w[n]$ is unique in w, that is, $w[n] \neq w[i]$ for any $1 \leq i \leq n-1$. Then, for any string $x \in \mathit{Factor}(w)$, $\overset{w}{\overrightarrow{x}} = \overset{w}{\overset{\Longrightarrow}{x}}$.*

Proof. By the contraposition of the second statement of Lemma 2, if $x \notin \mathit{Suffix}(w)$, then $\overset{w}{\overrightarrow{x}} = \overset{w}{\overset{\Longrightarrow}{x}}$. Because of the unique character $w[n]$, any suffix z of w satisfies the precondition of Lemma 3, and thus $\overset{w}{\overrightarrow{z}} = \overset{w}{\overset{\Longrightarrow}{z}}$. □

We are now ready to define $\mathit{STree}'(w)$, which is a modified version of $\mathit{STree}(w)$.

Definition 2 *$\mathit{STree}'(w)$ is the tree (V, E) such that*

$$V = \{\overset{w}{\overset{\Longrightarrow}{x}} \mid x \in \mathit{Factor}(w)\},$$

$$E = \{(\overset{w}{\overset{\Longrightarrow}{x}}, a\beta, \overset{w}{\overset{\Longrightarrow}{xa}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma,\ \beta \in \Sigma^*,\ \overset{w}{\overset{\Longrightarrow}{xa}} = xa\beta, \text{ and } \overset{w}{\overset{\Longrightarrow}{x}} \neq \overset{w}{\overset{\Longrightarrow}{xa}}\},$$

and its suffix links are the set

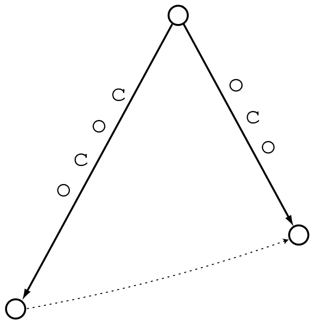
$$F = \{(\overset{w}{\overset{\Longrightarrow}{ax}}, \overset{w}{\overset{\Longrightarrow}{x}}) \mid x, xa \in \mathit{Factor}(w),\ a \in \Sigma, \text{ and } \overset{w}{\overset{\Longrightarrow}{ax}} = a \cdot \overset{w}{\overset{\Longrightarrow}{x}}\}.$$

Remark that $STree'(w)$ can be obtained by replacing $\overset{w}{\overrightarrow{(\cdot)}}$ in $STree(w)$ with $\overset{w}{\Longrightarrow}(\cdot)$.

We have the next lemma deriving from Lemma 4.

Lemma 5 *Let $w \in \Sigma^*$ with $|w| = n$. Assume that the last character $w[n]$ is unique in w, that is, $w[n] \neq w[i]$ for any $1 \leq i \leq n-1$. Then, $STree(w) = STree'(w)$.*

For comparing $STree(w)$ and $STree'(w)$, see Figure 1 and Figure 2. As shown in Proposition 3, any suffixes of a string represented by an explicit node are also explicit.



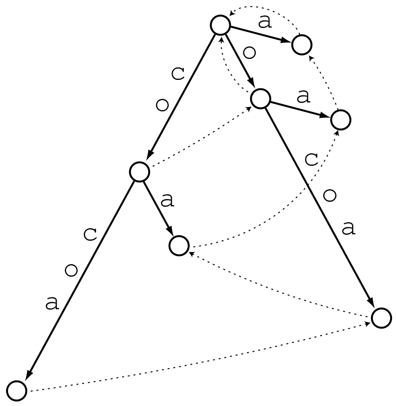


Figure 2: $STree'$(`coco`) on the left, and $STree'$(`cocoa`) on the right. Solid arrows represent the edges, while dotted arrows denote suffix links.

According to Lemma 5, using a delimiter \$ that occurs nowhere in w, we have $STree(w\$) = STree'(w\$)$ for any $w \in \Sigma^*$.

3 Bidirectional Construction of Suffix Trees

3.1 Right Extension

Assume that we have $STree'(w)$ with some $w \in \Sigma^*$. Now we consider updating it into $STree'(wa)$ with $a \in \Sigma$, by inserting the suffixes of wa into $STree'(w)$. Ukkonen [22] achieved the following result.

Theorem 2 ([22]) *For any $a \in \Sigma$ and $w \in \Sigma^*$, $STree'(w)$ can be updated to $STree'(wa)$ in amortized constant time.*

Here we only recall essence of Ukkonen's algorithm together with some supporting lemmas and propositions.

Let y be the longest string in $Factor(w) \cap Suffix(wa)$. Then y is called the *longest repeated suffix* of wa and denoted by $LRS(wa)$. Since every string $x \in Suffix(y)$ belongs to $Factor(w)$, we do not need to newly insert any x into $STree'(w)$.

Lemma 6 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = LRS(w)$. For any string $x \in Suffix(w) - Suffix(y)$, $\overset{wa}{\overrightarrow{x}} = \overset{w}{\overrightarrow{x}} \cdot a$.*

Proof. Since $y = LRS(w)$, any string $x \in \mathit{Suffix}(w) - \mathit{Suffix}(y)$ appears only once in w as a suffix of w, and is therefore $\overset{w}{\overrightarrow{x}} = x$. Also, x is followed only by a in wa, and thus $\overset{wa}{\overrightarrow{x}} = xa$. □

This lemma implies that a leaf node of $STree'(w)$ is also a leaf node in $STree'(wa)$. Thus we need no explicit maintenance for leaf nodes. Namely, we can insert all strings of $\mathit{Suffix}(w) - \mathit{Suffix}(y)$ into $STree'(w)$ *automatically* (for more detail, see [22]).

Proposition 7 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = LRS(w)$ and $z = LRS(wa)$. For any string $x \in \mathit{Suffix}(y) - \mathit{Suffix}(z)a^{-1}$, $\overset{wa}{\overrightarrow{x}} = x$.*

Proof. Firstly, we consider the empty string ε. It always belongs to $\mathit{Suffix}(y) - \mathit{Suffix}(z)a^{-1}$, since $\varepsilon \in \mathit{Suffix}(y)$ and $\varepsilon \notin \mathit{Suffix}(z)a^{-1}$. It is now obvious that $\overset{wa}{\overrightarrow{\varepsilon}} = \varepsilon$. Now we consider other strings. That $xa \notin \mathit{Suffix}(z)$ implies the existence of $b \in \Sigma$ such that $xb \in \mathit{Factor}(w)$ and $b \neq a$. Therefore, we have $\overset{wa}{\overrightarrow{x}} = x$. □

We start from the location corresponding to $LRS(w)$ and convert $STree'(w)$ to $STree'(wa)$, while creating new explicit nodes if necessary to insert new suffixes into $STree'(w)$, according to the above proposition. Now the next question is how to detect the locations where new explicit nodes should be created.

We here define the *eliminator* ξ for any character $a \in \Sigma$ by

$$a\xi = \xi a = \varepsilon$$

and $|\xi| = -1$. Moreover, we define that $\xi \in \mathit{Prefix}(\varepsilon)$ and $\xi \in \mathit{Suffix}(\varepsilon)$, but $\xi \notin \mathit{Prefix}(x)$ and $\xi \notin \mathit{Suffix}(x)$ for any $x \in \Sigma^+$. The symbol ξ corresponds to the auxiliary node $\perp$ introduced by Ukkonen [22]. Owing to the introduction of ξ, we can establish the following lemma.

Lemma 7 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = LRS(w)$ and $z = LRS(wa)$. Let $x \in \mathit{Suffix}(y) - \mathit{Suffix}(z)a^{-1}$. Suppose t is the longest string in $\mathit{Prefix}(x)$ such that $\overset{w}{\overrightarrow{t}} = t$. Let $x' = \mathit{Suffix}(x)$ with $|x'| + 1 = |x|$ and $t' = \mathit{Suffix}(t)$ with $|t'| + 1 = |t|$. For string $\alpha \in \Sigma^*$ such that $t\alpha = x$, $t'\alpha = x'$.*

Notice that we can reach string x' via the suffix link of the node for t in $STree'(w)$ and along the path spelling out α from the node for t' (recall Definition 2). Moreover, Proposition 6 guarantees that t' is an explicit node in $STree'(w)$. Ukkonen proved that x' can be found in amortized constant time by using the suffix link of node $\overset{w}{\overrightarrow{t}}$.

3.2 Left Extension

Weiner [23] proposed an algorithm to construct $STree(aw)$ by updating $STree(w)$ with $a \in \Sigma$ in amortized constant time. On the other hand, this section is devoted to the exposition of the conversion from $STree'(w)$ to $STree'(aw)$. In so doing, we insert prefixes of aw into $STree'(w)$.

Lemma 8 *Let $a \in \Sigma$ and $w \in \Sigma^*$. For any string $x \in Factor(w) - Prefix(aw)$, $\overset{w}{\overset{\Longrightarrow}{x}} = \overset{aw}{\overset{\Longrightarrow}{x}}$.*

Proof. Let b be the unique character that follows x in w. (When $\overset{w}{\overset{\Longrightarrow}{x}} = x$, then $b = \varepsilon$.) Since $x \notin Prefix(aw)$, there is no new occurrence of x in aw. Therefore, b is also the only character following x in aw. Hence $\overset{w}{\overset{\Longrightarrow}{x}} = \overset{aw}{\overset{\Longrightarrow}{x}}$. □

The above lemma ensures that any implicit node of $STree'(w)$ does not become explicit in $STree'(aw)$ if it is not associated with any prefix of aw.

Now we turn our attention to the strings in $Prefix(aw)$. Let x be the longest string in set $Factor(w) \cap Prefix(aw)$. Then x is called the *longest repeated prefix* of aw and denoted by $LRP(aw)$. Since all prefixes of x belong to $Factor(w)$, we need not newly insert any of them into $STree'(w)$.

Proposition 8 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $x = LRP(aw)$ and $y = LRS(w)$. If $x \notin Suffix(w) - Suffix(y)$, then $\overset{aw}{\overset{\Longrightarrow}{x}} = x$. Otherwise, $\overset{aw}{\overset{\Longrightarrow}{x}} = aw$.*

Proof. We first consider the case that $x \notin Suffix(w) - Suffix(y)$. Recall that x is the *longest* string in $Factor(w) \cap Prefix(aw)$. Moreover, $x \notin Suffix(w) - Suffix(y)$. Hence, there exist two characters $b, c \in \Sigma$ such that $xb, xc \in Factor(aw)$ and $b \neq c$. Thus we have $\overset{aw}{\overset{\Longrightarrow}{x}} = x$.

Now we consider the second case, $x \in Suffix(w) - Suffix(y)$. Here, x occurs only once in w as its suffix. Thus $\overset{w}{\overset{\Longrightarrow}{x}} = x$. On the other hand, by the definition of $LRP(aw)$, we obtain $x \in Prefix(aw) - \{aw\}$. Therefore, there uniquely exists a character $d \in \Sigma$ which follows x in aw. Hence we have $\overset{aw}{\overset{\Longrightarrow}{x}} = aw$. □

The above proposition implies that if $LRP(aw)$ is not on a leaf node in $STree'(w)$, it is represented by an explicit node in $STree'(aw)$, and otherwise it becomes implicit in $STree'(aw)$. We stress that this characterizes a difference between $STree'(w)$ and $STree(w)$. More concretely, Weiner's original algorithm constructs $STree(aw)$ on the basis of the next proposition.

Proposition 9 *For any $a \in \Sigma$ and $w \in \Sigma^*$, if $x = LRP(aw)$, then $\overset{aw}{\overrightarrow{x}} = x$.*

Now the next question is how to locate $LRP(aw)$ in $STree'(w)$. Our idea is similar to Weiner's strategy for constructing $STree(w)$ [23]. Let y be the longest element in set $Prefix(w) \cup \{\xi\}$ such that $ay \in Factor(w)$. Then y is called the *base* of aw and denoted by $Base(aw)$. On the other hand, let z be the longest element in set $Prefix(w) \cup \{\xi\}$ such that $\overset{w}{\overset{\Longrightarrow}{az}} = az$. Then z is called the *bridge* of aw and denoted by $Bridge(aw)$.

Lemma 9 ([23]) *Let $a \in \Sigma$ and $w \in \Sigma^*$. If $y = Base(aw)$, then $ay = LRP(aw)$.*

Proof. Assume contrarily that y' is the string such that $ay' = LRP(aw)$ and $|y'| > |y|$. By the definition of $LRP(aw)$, we have $ay' \in Prefix(aw)$, which yields $y' \in Prefix(w)$. It, however, contradicts the precondition that $y = Base(aw)$ since $|y'| > |y|$. □

According to the above lemma, we can utilize $Base(aw)$ for finding $LRP(aw)$ in $STree'(w)$.

Lemma 10 *Let $a \in \Sigma$ and $w \in \Sigma^*$. If $x = LRP(w)$, $y = Base(aw)$ and $z = Bridge(aw)$, then $y \in Prefix(x)$ and $z \in Prefix(y)$.*

Proof. By Lemma 9 we have $ay = LRP(aw)$. It is easy to see that $|LRP(w)| + 1 \geq |LRP(aw)|$, which implies $|x| \geq |y|$. Since $x, y \in Prefix(w)$, we obtain $y \in Prefix(x)$. It can be readily shown that $az \in Prefix(ay)$, since $ay = LRP(aw)$. Thus we have $z \in Prefix(y)$. □

The above lemma ensures that we can find both $Base(aw)$ and $Bridge(aw)$ by going up along the path from the node of $LRP(w)$ in $STree'(w)$.

Lemma 11 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Let $y = Base(aw)$ and $z = Bridge(aw)$. Assume $\gamma \in \Sigma^*$ is the string satisfying $z\gamma = y$. Then, $az\gamma = LRP(aw)$.*

Proof. By Lemma 9 and Lemma 10. □

According to the above lemma, we can locate $LRP(aw)$ in $STree'(w)$ by going down from the node $\overset{w}{\overrightarrow{az}}$. The only thing not clarified yet is how to move from node $\overset{w}{\overrightarrow{z}}$ to node $\overset{w}{\overrightarrow{az}}$. If we maintain the set F' below, we can detect $LRP(aw)$ in constant time, where

$$F' = \{(\overset{w}{\overrightarrow{x}}, a, \overset{w}{\overrightarrow{ax}}) \mid x, ax \in Factor(w), a \in \Sigma, \text{ and } \overset{w}{\overrightarrow{ax}} = a \cdot \overset{w}{\overrightarrow{x}}\}.$$

Comparing F' and F in Definition 2, one can see that F' is the set of the *labeled reversed suffix links* of $STree'(w)$.

We now have the following theorem.

Theorem 3 *For any $a \in \Sigma$ and $w \in \Sigma^*$, $STree'(w)$ can be updated to $STree'(aw)$ in amortized constant time.*

3.3 Mutual Influences

Here, we consider mutual influences between Left Extension and Right Extension. The next lemma shows what happens to $LRP(w)$ when $STree'(w)$ is updated to $STree'(wa)$.

Lemma 12 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Assume $LRP(w) = LRS(w)$. Let $x = LRS(w)$. If $xa \in Prefix(w)$, then $LRP(wa) = xa$.*

Proof. Since $xa \in Prefix(w)$, $LRS(wa) = xa$. Thus $xa = LRP(wa)$. □

This lemma shows when and where $LRP(wa)$ moves from the location of $LRP(w)$ according to the character a newly added to the right of w. Examining the precondition, "if $xa \in Prefix(w)$", is feasible in $O(|\Sigma|)$ time, which regarded as $O(1)$ if Σ is a fixed alphabet.

The following lemma stands in contrast to Lemma 12.

Lemma 13 *Let $a \in \Sigma$ and $w \in \Sigma^*$. Assume $LRP(w) = LRS(w)$. Let $x = LRP(w)$. If $ax \in Suffix(w)$, then $LRS(aw) = ax$.*

This lemma shows when and where $LRS(aw)$ moves from the location of $LRS(w)$ according to the character a newly added to the left of w. Examining the precondition, "if $ax \in Suffix(w)$", is also feasible in $O(|\Sigma|)$ time, and moving from the location of $LRS(w)$ to that of $LRS(aw)$ can be done in constant time by the use of the labeled reversed suffix link of $LRP(w)$.

As a result of discussion, we finally obtain the following:

Theorem 4 *For any string $w \in \Sigma^*$, $STree'(w)$ can be constructed in bidirectional manner and in $O(|w|)$ time.*

A bidirectional construction of $STree'(w)$ with $w =$ `cocoon` is displayed in Figure 3.

4 Concluding Remarks

We introduced an algorithm for bidirectional construction of suffix trees, which performs in linear time. It should be noted that the proposed algorithm can construct an index of w^{rev} at the same time, where w^{rev} is the reversal of a given string w. In [14], we improved Ukkonen's algorithm so as to construct not only $STree'(w)$ but also $DAWG(w^{\mathrm{rev}})$ in right-to-left on-line manner. The algorithm of this paper leads bidirectional construction of $STree'(w)$ and $DAWG(w^{\mathrm{rev}})$, although theoretical details are omitted in this draft.

Acknowledgment

The author wishes to thank Prof. Ayumi Shinohara and Prof. Masayuki Takeda. Daily fruitful and enthusiastic discussion with them led the author to the inspiration for this work.

References

[1] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithm on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 85–96. Springer-Verlag, 1985.

[2] M. Balík. Implementation of dawg. In *Proc. The Prague Stringology Club Workshop '98 (PSCW'98)*. Czech Technical University, 1998.

[3] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[4] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *J. ACM*, 34(3):578–595, 1987.

[5] D. Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, 191:131–144, 1998.

[6] M. T. Chen and J. Seiferas. Efficient and elegant subword tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithm on Words*, volume 12 of *NATO Advanced Science Institutes, Series F*, pages 97–107. Springer-Verlag, 1985.

[7] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

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

[9] M. Crochemore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *Lecture Notes in Computer Science*, pages 192–211. Springer-Verlag, 1997.

[10] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. The 38th Annual Symposium on Foundations of Computer Science (FOCS '97).* IEEE Computer Society, 1997.

[11] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

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

[13] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. Construction of the CDAWG for a trie. In *Proc. The Prague Stringology Conference '01 (PSC'01).* Czech Technical University, 2001.

[14] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. On-line construction of symmetric compact directed acyclic word graphs. In *Proc. of 8th International Symposium on String Processing and Information Retrieval (SPIRE'01)*, pages 96–110. IEEE Computer Society, 2001.

[15] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. In A. Amir and G. M. Landau, editors, *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 169–180. Springer-Verlag, 2001.

[16] S. Inenaga, A. Shinohara, M. Takeda, and S. Arikawa. Compact directed acyclic graphs for a sliding window. In *Proc. of 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002. (to appear).

[17] M. G. Maaß. Linear bidirectional on-line construction of affix trees. In R. Giancarlo and D. Sankoff, editors, *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, volume 1848 of *Lecture Notes in Computer Science*, pages 320–334. Springer-Verlag, 2000.

[18] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Compt.*, 22(5):935–948, 1993.

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

[20] J. Stoye. Affixbäume. Master's thesis, Universität Bielefeld, 1995. (in German).

[21] J. Stoye. Affix trees. Technical Report 2000–4, Universität Bielefeld, Technische Fakultät, 2000.

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

[23] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

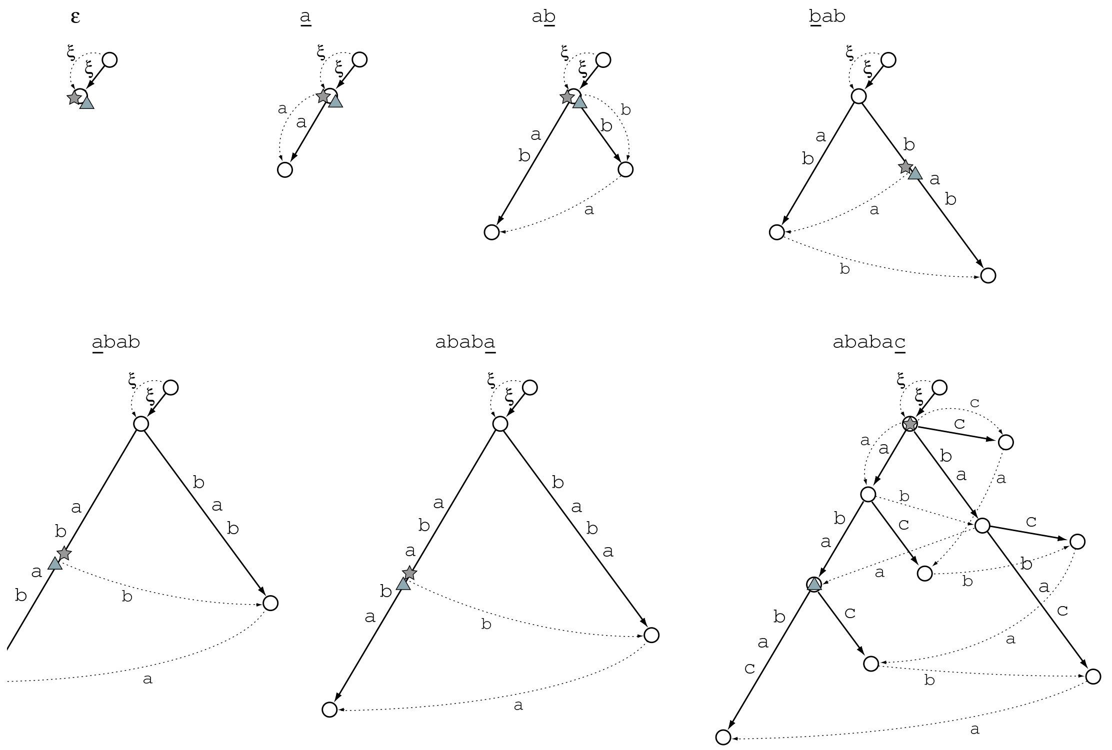


Figure 3: A bidirectional construction of $\mathit{STree}'(w)$ with $w = \mathtt{ababac}$. Solid arrows represent edges while dotted arrows denote labeled reversed suffix links. On Right Extension, labeled reversed suffix links are used for the reversed direction, that is, as "normal" suffix links. In each phase, a gray triangle (star, respectively) indicates the location of the longest repeated prefix (suffix, respectively). The newly added character is underlined in each phase. When $\mathit{STree}'(\mathtt{ab})$ is updated to $\mathit{STree}'(\mathtt{bab})$, the node for string `b` becomes implicit (Proposition 8). Due to the conversion of $\mathit{STree}'(\mathtt{bab})$ into $\mathit{STree}'(\mathtt{abab})$, $\mathit{LRP}(\mathtt{abab})$ moves via the labeled reversed suffix link, and $\mathit{LRS}(\mathtt{abab})$ also moves to the same position according to Lemma 13. Then, the suffix tree is updated to $\mathit{STree}'(\mathtt{ababa})$ where $\mathit{LRS}(\mathtt{ababa})$ moves while spelling out the new character `a` along the edge. Note that $\mathit{LRP}(\mathtt{ababa})$ also moves due to Lemma 12. Since the precondition of Lemma 12 is not satisfied in the string `ababac`, $\mathit{LRP}(\mathtt{ababac})$ does not move in $\mathit{STree}'(\mathtt{ababac})$. For smart construction, we also maintain the labeled reversed suffix link of the longest repeated suffix even if it is *not* on an explicit node (see $\mathit{STree}'(\mathtt{bab})$, for instance). This labeled reversed suffix link is the only suffix link that would be "modified" after it is created. For example, the labeled reversed suffix link of the node for string `a` in $\mathit{STree}'(\mathtt{a})$ is deleted in $\mathit{STree}'(\mathtt{ab})$ since it no longer satisfies the definition of labeled reversed suffix links. On the other hand, that of the node for string `ab` in $\mathit{STree}'(\mathtt{abab})$ still exists in $\mathit{STree}'(\mathtt{ababa})$ as that of the node for string `aba`.

Image Recognition Using Finite Automata

Tomáš Skopal, Václav Snášel, Michal Krátký

Department of Computer Science
VŠB-Technical University Ostrava
17. listopadu 15, 708 33 Ostrava
Czech Republic

e-mail: {tomas.skopal, vaclav.snasel, michal.kratky}@vsb.cz

Abstract. In this paper we introduce an idea of image recognition using conventional (single-dimensional) finite automata. This approach could be an elegant alternative to complicated solutions based on two-dimensional languages and two-dimensional automata. In consequence, this method could be generally extended to the context of higher-dimensional languages beyond the scope of image recognition.

1 Introduction

Image recognition recently became an object of interest for theory of automata. The picture, a rectangular raster, can be considered as a sentence of a two-dimensional language where the pixels of picture are characters of a finite alphabet.

It is obvious that sentences of two or more-dimensional language cannot be recognized by "conventional" automata. Conventional automaton takes the characters from the input one by one as they appear in a single-dimension sentence. On the other side, two-dimensional sentence processing (e.g. picture) is not so unambiguous, there exist four directions in which the sentence can be processed in each step – *left, right, upwards, downwards*. There were several two-dimensional automata designed, e.g. *4-way* finite-state automata [BH67].

Our solution tries to exploit the existing well-established area of "conventional" automata together with the transformation of the two-dimensional language into a single-dimensional one. The transformation of a picture (or two-dimensional sentence) consists of *space linearization*. This means that the pixels of a picture are linearly ordered and the resultant ordering along with the original picture define the appropriate single-dimensional sentence. The linear order is performed using a space filling curve. In this paper we propose certain curves which were proved to be the good space-describing curves in many applications (especially in data storage and retrieval). However, the quality of the curves may differ in our case and therefore we refer to [SKS02] where we discuss some general properties of space filling curves.

Once we have chosen the curve for language description we must construct an automaton that recognizes a given picture in its "flat shape". However, none of the space filling curves describe the space (and picture) perfectly, some distortion of the picture recognition must be taken into account. This seeming drawback can turn over to an advantage if we realize that the measure of recognition distortion may represent *similarity* of the recognized picture to the prospective pattern.

Automaton construction for recognition of the linearized picture is based on the Levenshtein DFA where the Levenshtein metric (edit distance) serves as the measure for the allowed picture distortion.

2 Two-dimensional Languages

Informally, a two-dimensional string is called a picture and is defined as a rectangular array of symbols taken from finite alphabet Σ. A two-dimensional language (e.g. picture language) is then a set of pictures.

A generalization of formal languages to two dimensions is possible in different ways, and several formal models to recognize or generate two-dimensional objects have been proposed in the literature (see [KM1, KM2, LMN98]). These approaches were initially motivated by problems arising in the framework of pattern recognition and image processing.

Definition [RS97] A two-dimensional string (e.g. picture) over Σ is a two-dimensional rectangular array of elements from Σ. The set of all two-dimensional strings over Σ is denoted as Σ^{**}. A two-dimensional language over Σ is a subset of Σ^{**}.

Given a picture $p \in \Sigma^{**}$, $l_1(p)$ denotes the number of rows and $l_2(p)$ denotes the number of columns of p.

The pair $(l_1(p), l_2(p))$ is called the size of the picture p. The set of all pictures over Σ of size (m, n), with $m, n > 0$ will be indicated as $\Sigma^{m\times n}$. Furthermore, if $1 \leq i \leq l_1(p)$ and $1 \leq j \leq l_2(p)$, then $p(i, j)$ (or equivalently $p_{i,j}$) denotes the symbol in picture p on coordinates (i, j).

Two-dimensional languages, or picture languages, are an interesting generalization of the standard languages of computer science. Rather than one-dimensional strings, we consider two-dimensional arrays of symbols over a finite alphabet. These arrays can then be accepted or rejected by various types of automata. The introduction of two-dimensional automata brought a new sort of automata on the stage, with its own huge theoretical background.

3 Another Approach

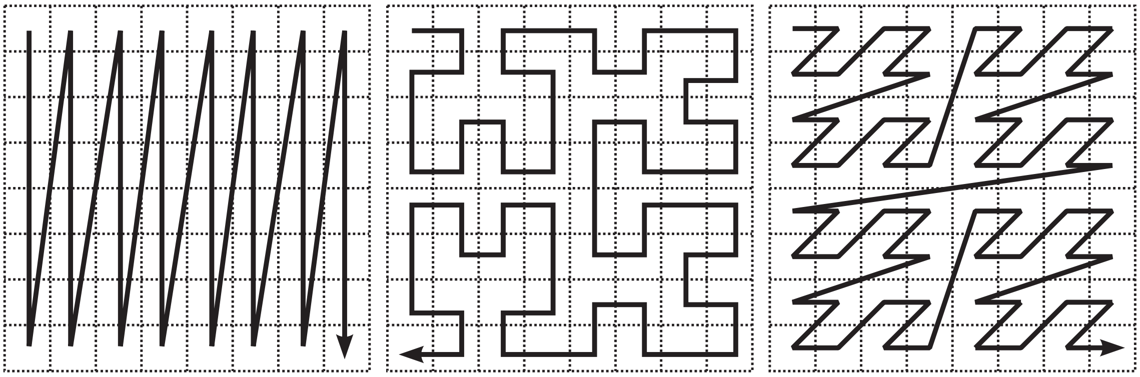
Our approach is to reuse the existing traditional (single-dimensional) automata (languages respectively) and simplify the automaton construction problem. The most important thing is to transform the two-dimensional language (pictures) into one-dimensional strings. This can be done using space filling curves. The consecutive automaton construction depends on the properties of space filling curve we have chosen.

3.1 Space Filling Curves

We want to transform the *two-dimensional string* over Σ into the *one-dimensional string* over Σ. The two-dimensional string over Σ is a two-dimensional rectangle array of elements of Σ. We can look at the array as a two dimensional space $\Omega = D_1 \times D_2$, where the cardinality of domain D_1 ($|D_1|$) is equal to the rows count of the array and $|D_2|$ is equal to the columns count. The tuple (point) with coordinates $(column, row)$ within the space will have a value in Σ.

Many space filling curves have been developed, for example *C-curve*, *Z-curve* or *Hilbert curve* ([Ma99]). For deeper acquaintance with the topic of general space filling curves we refer to the comprehensive monography by Hans Sagan [Sa94].

The usage of the curves isn't in two-dimensional space only, but the curves fill any vector space with arbitrary dimension. It is possible to use the curves for transformation of the n-dimensional string over Σ into the one-dimensional string over Σ. We can see C-curve, Hilbert curve, and Z-curve filling the two dimensional space 8×8 in Figure 1.



a) b) c)

Figure 1: The space filling curves. a) C-curve, b) Hilbert curve, and c) Z-curve.

We can consider several curves, but it is convenient to choose the curve that is highly *self-similar* ([Ma99], [SKS02]) – informally, it means that points that are geometrically close, would have to lie close on the curve. For example, the Z-curve is used for indexing of multidimensional data with UB-trees ([Ba97]). In the following section we will describe the Z-curve as an example of space filling curve.

3.2 Z-address

Definition 1 **(Z-address)**
Let Ω be an n-dimensional space. For tuple $\mathcal{O} \in \Omega$ with n attributes and binary representation attribute value $A_i = A_{i,s-1}A_{i,s-2}\dots A_{i,0}$, where $1 \le i \le n$. Then

$$Z(\mathcal{O}) = \sum_{j=0}^{s-1}\sum_{i=1}^{n} A_{i,j}2^{jn+i-1}$$

is the *Z-address* function for space Ω.

The attributes of tuple define the coordinates of point representing tuple in the space Ω. If we are calculating the Z-address for all points of n-dimensional space Ω and order the points according their Z-address value, we get the Z-curve filling the entire space Ω (see Figure 2a). For calculation of tuple Z-address exists algorithm with linear complexity - so called *bit interleaving algorithm* (see bellow).

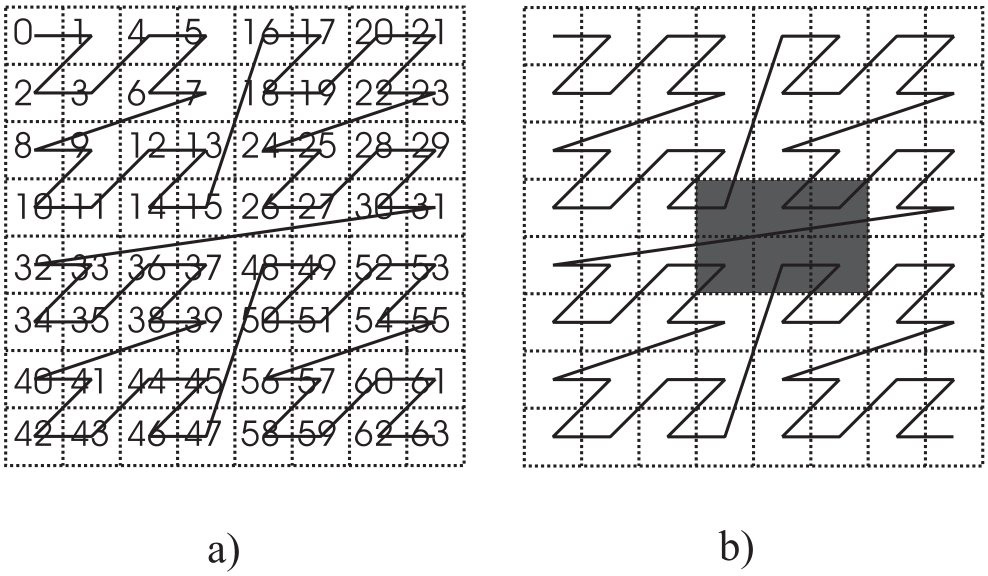


Figure 2: a) The two-dimensional space (image) 8×8 filled by Z-curve. b) Picture in image interleaved by the Z-curve.

Z-address calculation example

We see the calculation of Z-address according bit interleaving algorithm for point (6,13) in two-dimensional space in Figure 3. Numbers 6 and 13 have the binary form 0110 and 1101 respectively. We obtain the coordinate values as four places bit strings. Maximal values for four places binary number is 16. The domains D_i of both attribute are sets $\{0,1, \ldots, 14,15\}$, point lies in two-dimensional space 16×16. The result point Z-address is then 10110110 (182 decimal).

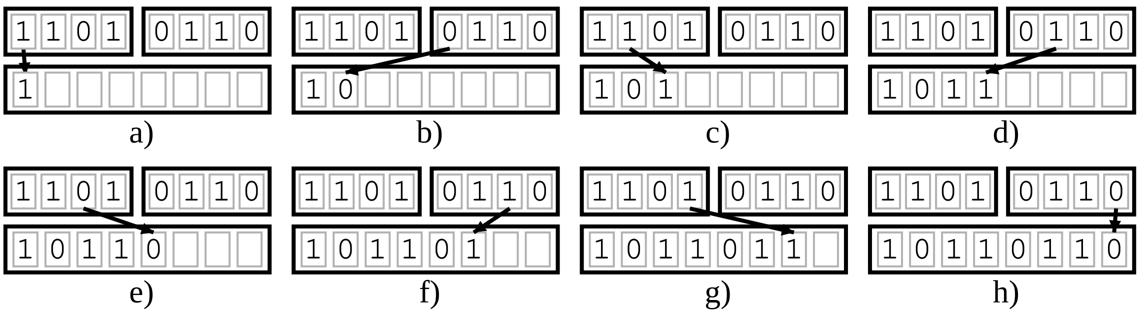


Figure 3: The Z-address calculation according to bit interleaving algorithm for point $(6, 13)$ in two-dimensional space 16×16.

It is possible to go through the entire space passing upon Z-curve. We interleave a picture (the two-dimensional string) over Σ by Z-curve and we recognize the picture by the "classical" one-dimensional finite automata (e.g. see Figure 2b). The automata construction of the picture recognition is outlined in the next section.

3.3 Automaton Construction

The automaton will recognize a square picture of size $x \times y$ within an image of a greater size (see Figure 2b).

Construction
The automaton type is well-known NDFA for matching patterns with k differences – in other words, it is an automaton for approximate string matching using Levenshtein metric. The construction takes as a parameter the pattern sentence (picture to be recognized) and a Levenshtein distance threshold which defines the maximal tolerance value of the above mentioned picture distortion. For detailed information on construction of the Levenshtein automata see [Ho96].

The Levenshtein distance threshold is computed as the minimal distance of the pattern picture to an input picture when the *correct* input is still recognizable. More clearly, the correct input picture may appear on any position in the image and the automaton must recognize the picture on this position. However, the threshold value may cause that they can be recognized also incorrect pictures. This imprecise behaviour could serve as a similarity recognition because the recognized picture is always within the Levenshtein distance threshold which guarantees only a limited number of differences between the pattern picture and the input picture. Pictures that are close (in terms of Levenshtein distance) could be considered as similar to each other.

In following we will focus on measuring of the pattern picture and input picture using Levenshtein metric.

3.3.1 What is the Levenshtein Distance?

Levenshtein distance (LD) is a measure of the similarity between two strings, which we will refer to as the source string (s) and the target string (t). The distance is the number of deletions, insertions, or substitutions required to transform s into t. For example,

If s is `"test"` and t is `"test"`, then $\mathrm{LD}(s,t) = 0$, because no transformations are needed. The strings are already identical.

If s is `"test"` and t is `"tent"`, then $\mathrm{LD}(s,t) = 1$, because one substitution (change "s" to "n") is sufficient to transform s into t. The greater the Levenshtein distance, the more different the strings are.

Levenshtein distance is named after the Russian scientist Vladimir Levenshtein, who devised the algorithm in 1965 [Le66]. If you can't spell or pronounce Levenshtein, the metric is also sometimes called *edit distance*.

The Levenshtein distance algorithm (based on dynamic programming) has been used in:

- Spell checking
- Speech recognition

- DNA analysis
- Plagiarism detection

The Algorithm – step description

1. Set n to be the length of s.
   Set m to be the length of t.
   If $n = 0$, return m and exit.
   If $m = 0$, return n and exit.
   Construct a matrix containing $0 \dots m$ rows and $0 \dots n$ columns.
2. Initialize the first row to $0 \dots n$.
   Initialize the first column to $0 \dots m$.
3. Examine each character of s (i from 1 to n).
4. Examine each character of t (j from 1 to m).
5. If $s[i]$ equals $t[j]$, the cost is 0.
   If $s[i]$ doesn't equal $t[j]$, the cost is 1.
6. Set cell d[i,j] of the matrix equal to the minimum of: a. The cell immediately above plus 1: $d[i-1, j] + 1$.
   b. The cell immediately to the left plus 1: $d[i, j-1] + 1$.
   c. The cell diagonally above and to the left plus the cost: $d[i-1, j-1] + cost$.
7. After the iteration steps (3, 4, 5, 6) are complete, the distance is found in cell $d[n, m]$.

3.4 Examples

As we have said earlier, the Levenshtein threshold value is computed as a maximum distance of the pattern picture and the *correct* input picture on any position in the image being recognized. In Figure 4 are depicted three examples of pictures (sized 3×3) in images (sized 8×8) and its distances to pattern pictures.

Note that the pixel values are characters from a finite alphabet. The numbers next to the pixels are the character identifiers. The gaps denoting those pixels of image that are not pixels of the picture are represented with appropriate characters but in our examples, for simplicity and clarity, the gap is represented with a special character that is not contained in the alphabet Σ. This special character ensures the worst matching case, thus the real distance computations will be always smaller or equal.

3.5 Extension to Multidimensional Languages

Because the space filling curve remains single-dimensional even for multidimensional spaces, we can extend the scope of two-dimensional languages to the multidimensional languages without the need of changing the automaton construction. Then, multidimensional sentences can be constructed simply by extending the language with additional coordinates.

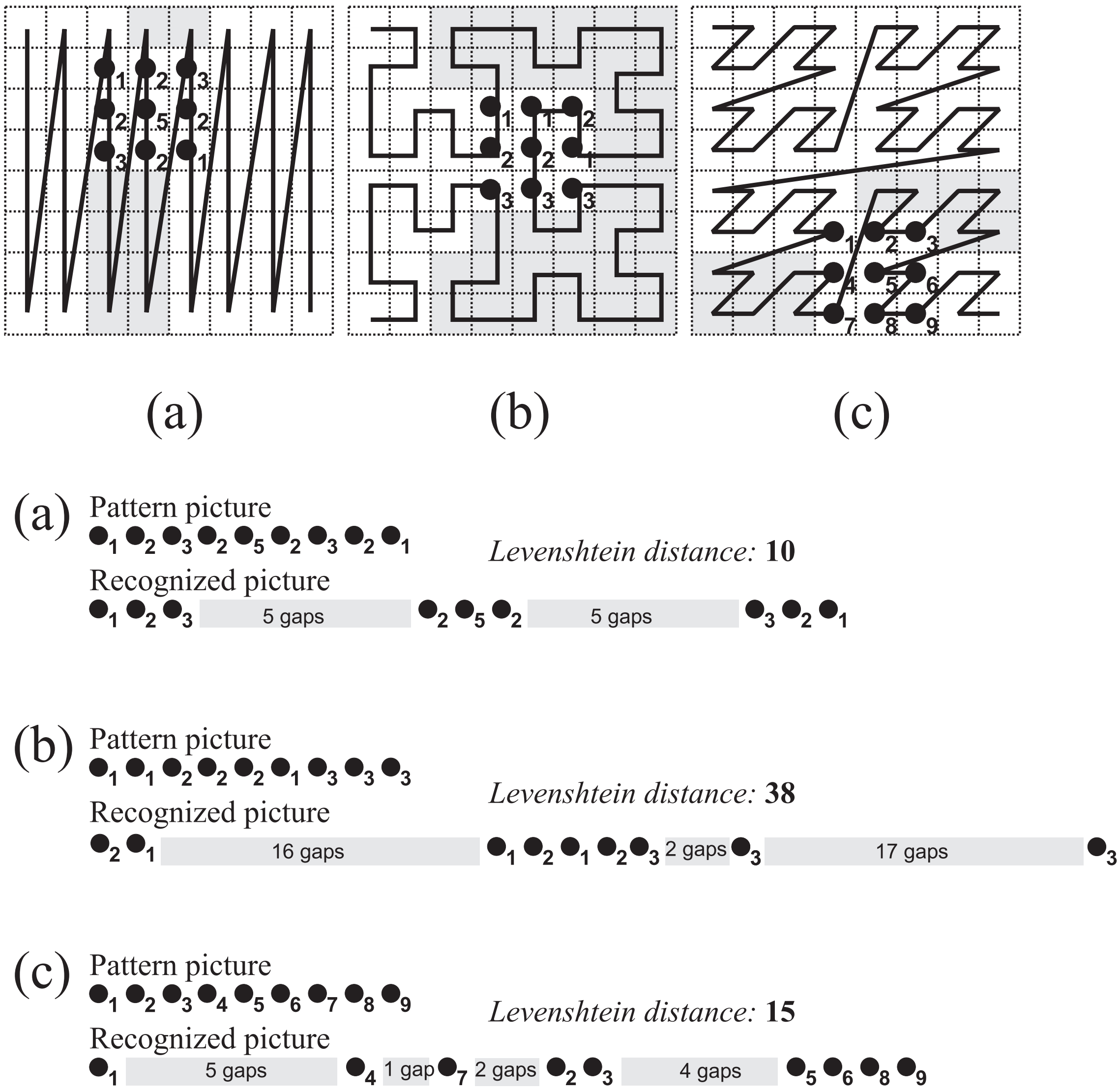


Figure 4: Measuring the Levenshtein distance on pictures.

In general, we can say that the imprecision caused by the Levenshtein distance threshold will increase with increasing dimension. This fact arises from the behaviour of the space filling curves in high-dimensional vector spaces. The other factor is the relation of sentence size to space size. The longer sentences and smaller sentence/space size ratio, the lower imprecision.

4 Conclusions

In this paper we have proposed an alternative solution of image recognition and even multidimensional language recognition. This method is based on space filling curves and Levenshtein automaton construction. The interesting property of this approach is an ability of similarity recognition.

References

[Ba97] Bayer R. The Universal B-Tree for multidimensional indexing: General Concepts. In: *Proc. Of World-Wide Computing and its Applications 97 (WWCA 97)*. Tsukuba, Japan, 1997.

[BH67] M.Blum, C.Hewitt. Automata on a 2-dimensional tape, *8th IEEE Symp. on Switching and Automata Theory*, 1967, pp. 155-160

[Ho96] J.Holub. Reduced Nondeterministic Finite Automata for Approximate String Matching, *Proceedings of the Prague Stringologic Club Workshop*, 1996

[KM1] J.Kari, C.Moore. Rectangles and Squares Recognized by Two-dimensional Automata, submitted, 2002

[KM2] J.Kari, C.Moore. New results on alternating and non-deterministic two-dimensional finite-state automata, In: *Proc. of the Symposium on Theoretical Aspects of Computer Science (STACS)*, LNCS, 2001.

[LMN98] K.Lindgren, C.Moore, M.Nordahl. Complexity of Two-dimensional Patterns, In: *Journal of Statistical Physics* 91(5-6) (1998) 909-951.

[RS97] D. Giammarresi, A. Restivo. Two-Dimensional Languages, In: *Handbook of Formal Languages*, vol 3, G. Rowzenberg and A. Salomaa eds, Springer-Verlag, 1997, chapter 4, 215–267.

[Le66] V.I.Levenshtein. Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics-Doklady 10* (1966), 707-710.

[Ma99] Markl, V.: *Mistral: Processing Relational Queries using a Multidimensional Access Technique*, Ph.D. thesis, Technical University Munchen, `http://mistral.in.tum.de/results/publications/Mar99.pdf`, 1999

[Sa94] Sagan H. *Space-Filling Curves*, Springer-Verlag, 1994

[SKS02] Skopal T., Krátký M., Snášel V.: Properties of Space Filling Curves And Usage With UB-trees. Submitted to MIS 2002.

Split and join for minimizing: Brzozowski's algorithm

J.-M. Champarnaud[1], A. Khorsi[2], T. Paranthoën[1]

[1] LIFAR, University of Rouen, France
{champarnaud,paranthoen}@dir.univ-rouen.fr

[2] CSD, University of Djilali Liabes, Sidi-Bel-Abbes, Algeria
ahmed_khorsi@lycos.com

Abstract. Brzozowski's minimization algorithm is based on two successive determinization operations. There is a paradox between its (worst case) exponential complexity and its exceptionally good performance in practice. Our aim is to analyze the way the twofold determinization performs the minimization of a deterministic automaton. We give a characterization of the equivalence classes of $\mathcal{A}$ w.r.t. the set of states of the automaton computed by the first determinization. The second determinization is expected to compute these equivalence classes. We show that it can be replaced by a specific procedure based on the classes characterization, which leads to a more efficient variant of Brzozowski's algorithm.

Key words: Finite automata, DFA minimization, Brzozowski's algorithm.

1 Introduction

It is well known that given a regular language L over an alphabet Σ there exists a canonical deterministic automaton which recognizes L, namely the minimal (deterministic) automaton of L, whose states are the left quotients of L w.r.t. the words of Σ^*. This automaton, denoted by $\mathcal{A}_L$, is unique (up to an isomorphism) and it has a minimal number of states [13]. Moreover, it can be computed from any deterministic automaton recognizing L by merging states which have identical right languages. There exist numerous algorithms to minimize a deterministic automaton. Watson published a taxonomy on this topic [18].

Among the various possible constructions, Brzozowski's minimization algorithm [3] is of a specific interest, regarding to several criteria which are discussed below. Let us first recall how it works. Let $\mathcal{A}$ be a (non necessarily deterministic) automaton, $d(\mathcal{A})$ be the subset automaton of $\mathcal{A}$ and $r(\mathcal{A})$ be the reverse automaton of $\mathcal{A}$. Brzozowski's algorithm is based on the following theorem:

$$\mathcal{A}_L = d(r(d(r(\mathcal{A}))))$$

This is a deep result since it relates DFA minimization to a basic operation, the determinization one. Let us mention that it has been generalized by Mohri to the case of bideterminizable transducers defined on the tropical semiring [12]. Brzozowski's theorem is also a fundamental tool for the computation of the nondeterministic minimal automata of a regular language. Let us cite the implementation [6] of the canonical automaton $\mathcal{C}_L$ defined by Carrez [4, 1] and the construction of the fundamental automaton $\mathcal{F}_L$ by Matz and Potthoff [11].

We are here especially interested by algorithmic and complexity features. Watson used the fact that Brzozowski's algorithm can take a nondeterministic automaton as input to design an algorithm which directly constructs a minimal deterministic automaton from a regular expression [19]. Since our aim is to study the way Brzozowski's algorithm performs a minimization, we will essentially consider the case when the initial automaton is a deterministic one. The paradox is the following: since Brzozowski's algorithm performs two determinizations, its (worst case) complexity is exponential w.r.t. the number of states of the initial automaton; nevertheless, as reported by Watson [18], Brzozowski's algorithm has proved to be exceptionally good in practice, usually out-performing Hopcroft's algorithm [7] significantly. Let us add that the average complexity of the algorithm has been proved to be exponential for group automata, although they likely are a favourable case since they are both deterministic and codeterministic [14].

Our contribution is the following. Let $\mathcal{A}$ be a deterministic automaton. We give a characterization of the equivalence classes of $\mathcal{A}$ w.r.t. the set of states of $dr(\mathcal{A})$, that is after the first determinization. The second determinization is expected to compute these equivalence classes. We show it can be replaced by a specific procedure based on the classes characterization, which leads to a more efficient variant of Brzozowski's algorithm.

Next section recalls some useful notations and definitions of automata theory. Section 3 is especially devoted to determinization and minimization operations. Section 4 presents Brzozowski's minimization algorithm and its proof. Section 5 provides an original analysis of the algorithm and the variant it leads to.

2 Preliminaries

Let us first review basic notions and terminology concerning finite automata and regular languages. For further details, classical books [2, 8] or handbooks [20] are excellent references.

Let Σ be a non-empty finite set of *symbols*, called the *alphabet*. Symbols are denoted by $x_1, x_2, \ldots, x_m$. A *word* u over Σ is a finite sequence $(y_1, y_2, \ldots, y_n)$ of symbols, usually written $y_1 y_2 \ldots y_n$. The *length* of a word u, denoted $|u|$ is the number of symbols in u. The *empty word* denoted by ε has a zero length. If $u = y_1 y_2 \ldots y_n$ and $v = z_1 z_2 \ldots z_p$ are two words over Σ, their *concatenation* $u \cdot v$, usually written uv, is the word $y_1 y_2 \ldots y_n z_1 z_2 \ldots z_p$. The set of all the *words* over Σ is denoted Σ^*. A *language* over Σ is a subset of Σ^*. The operations of union, concatenation and star over the subsets of Σ^* are called *regular operations*. The *regular languages* over Σ are the languages obtained from the finite subsets of Σ^* by using a finite number of regular operations.

A (finite) automaton is a 5-tuple $\mathcal{M} = (Q, \Sigma, \delta, I, F)$ where Q is a (finite) set of states, Σ is a finite alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and δ is the transition function. The automaton $\mathcal{M}$ is *deterministic* ($\mathcal{M}$ is a DFA) if and only if $|I| = 1$ and δ is a mapping from $Q \times \Sigma$ to Q. Otherwise $\mathcal{M}$

is a *NFA* and δ is a mapping from $Q \times \Sigma$ to 2^Q. The automaton $\mathcal{M}$ is *complete* if and only if δ is a full mapping. A *path* of $\mathcal{M}$ is a sequence (q_i, a_i, q_{i+1}), $i = 1, \dots, n$, of consecutive edges. Its *label* is the word $w = a_1 a_2 \dots a_n$. A word $w = a_1 a_2 \dots a_n$ is *recognized* by the automaton $\mathcal{M}$ if there is a path with label w such that $q_1 \in I$ and $q_{n+1} \in F$. The language $L(\mathcal{M})$ *recognized* by the automaton $\mathcal{M}$ is the set of words which it recognizes. Two automata $\mathcal{M}$ and $\mathcal{M}'$ are *equivalent* if and only if they recognize the same language. A state is *accessible* (resp. *coaccessible*) if and only if there is a path from an initial state to this state (resp. from this state to a final state). An automaton is *trim* if and only if all its states are both accessible and coaccessible.

Kleene's theorem [10] states that a language is regular if and only if it is recognized by a finite automaton.

Let q be a state of $\mathcal{A} = (Q, \Sigma, \delta, i, F)$. The *right language* of q is the language $L_d^{\mathcal{A}}(q)$ (written $L_d(q)$ if not ambiguous) recognized by the automaton $\mathcal{A}_d(q) = (Q, \Sigma, \delta, q, F)$ obtained from $\mathcal{A}$ by making q the unique initial state. The *left language* of q is the language $L_g^{\mathcal{A}}(q)$ (written $L_g(q)$ if not ambiguous) recognized by the automaton $\mathcal{A}_g(q) = (Q, \Sigma, \delta, i, q)$ obtained from $\mathcal{A}$ by making q the unique final state. We will use the following proposition:

Proposition 1 *An automaton is deterministic if and only if the left languages of its states are pairwise disjoint.*

The *reverse* $r(u)$ *of the word* u is defined as follows: $r(\varepsilon) = \varepsilon$ and, if $u = u_1 u_2 \dots u_p$, then $r(u) = v_1 v_2 \dots v_p$, with $v_i = u_{p-i+1}$, for all i from 1 to p. The *reverse of the language* L is the language $r(L) = \{u \mid r(u) \in L\}$. The *reverse of the automaton* $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ is the automaton $r(\mathcal{A}) = (Q, \Sigma, r(\delta), F, I)$, obtained from $\mathcal{A}$ by swapping the role of initial and final states and by reversing the transitions.

We will use the following propositions, where $\mathcal{A}$ is a trim automaton:

Proposition 2 *If $\mathcal{A}$ recognizes the language L then $r(\mathcal{A})$ recognizes the language $r(L)$.*

Proposition 3 *If the left (resp. right) language of the state q in $\mathcal{A}$ is $L_g(q)$ (resp. $L_d(q)$), then its left (resp. right) language in $r(\mathcal{A})$ is $L_d(q)$ (resp. $L_g(q)$).*

3 Determinization and minimization operations

3.1 Determinization

Definition 1 *Let $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ be a NFA. The subset-automaton of $\mathcal{A}$ is the automaton $d(\mathcal{A}) = (Q', \Sigma, \delta', \{i'\}, F')$ defined as follows [8, 20]:*

- *Set of states: A deterministic state is a set of nondeterministic states; for all q' in Q', we have $q' \subseteq Q$.*
- *Initial state: The initial state in $d(\mathcal{A})$ is the set I of initial states in $\mathcal{A}$.*

- *Set of transitions: Let q' be a deterministic state and a be a symbol in Σ. If the transition from q' on symbol a is defined, then, by construction, its target is the state $\delta'(q', a)$ such that:*

$$\delta'(q', a) \quad = \quad \bigcup_{q \in q'} \delta(q, a). \tag{3}$$

- *Set of final states: A deterministic state is final if and only if it contains at least one final nondeterministic state: $q' \in F' \Leftrightarrow q' \cap F \neq \emptyset$.*

We will use the following proposition:

Proposition 4 *The right language of a state q' of $d(\mathcal{A})$ is equal to the union of the right languages of the states q of $\mathcal{A}$ belonging to the subset q'.*

Let n (resp. n') be the number of states in $\mathcal{A}$ (resp. in $d(\mathcal{A})$). As stated by Rabin and Scott [16], the upper bound $n' \leq 2^n - 1$ can be reached. Moreover, the automaton $d(\mathcal{A})$ can be computed with the following complexity [15, 5]: $O(\sqrt{n}2^{2n})$ when using lists, and $O(n^2(\log n)2^n)$ when using balanced search trees.

3.2 Minimization

The (left) quotient of a regular language L w.r.t. a word u of Σ^* is the language $u^{-1}L = \{v \in X^* \mid uv \in L\}$. The minimal automaton $\mathcal{A}_L$ of a regular language L is defined as follows:

- the set of states is the set of quotients of L,
- the initial state is L,
- the final states are the quotients which contain the empty word,
- the transition function is such that $\delta(u^{-1}L, x) = (ux)^{-1}L$.

The automaton $\mathcal{A}_L$ is unique up to an isomorphism and it has a minimal number of states [13]. We will use the following proposition:

Proposition 5 *A (deterministic, complete, accessible) automaton is minimal if and only if the right languages of its states are all different.*

The automaton $\mathcal{A}_L$ can be computed from any deterministic automaton recognizing L by merging states which are equivalent w.r.t. Nerode equivalence:

$$s \equiv t \Leftrightarrow [s \cdot u \in F \Leftrightarrow t \cdot u \in F, \, \forall u \in \Sigma^*]$$

Computing Nerode equivalence can be realized with a $O(n^2)$ complexity [13]. Using the notion of coarsest partition leads to a complexity of $O(nlog(n))$ [7].

4 Brzozowski's minimization algorithm

Let $\mathcal{A}$ be an automaton. Let $d(\mathcal{A})$ (resp. $r(\mathcal{A})$) be the subset automaton (resp. the reverse automaton) of $\mathcal{A}$. We will write $dr(\mathcal{A})$ for $d(r(\mathcal{A}))$, $rdr(\mathcal{A})$ for $r(d(r(\mathcal{A})))$ and $drdr(\mathcal{A})$ for $d(r(d(r(\mathcal{A}))))$.

Brzozowski's algorithm is based on the following theorem [3]:

Theorem 1 *(Brzozowski, 1962) Given a (non necessarily deterministic) automaton $\mathcal{A}$ recognizing a regular language L, the minimal deterministic automaton $\mathcal{A}_L$ of L can be computed by the formula:*

$$\mathcal{A}_L = drdr(\mathcal{A})$$

Proof. The proof is based on Propositions (1)–(5). By construction, the automaton $drdr(\mathcal{A})$ is deterministic, complete and accessible. From Proposition (2) it recognizes the language L. Let us show that the right languages of $drdr(\mathcal{A})$ are all distinct. From Proposition (1) the left languages of $dr(\mathcal{A})$ are pairwise disjoint. From Proposition (3) the right languages of $rdr(\mathcal{A})$ are the left languages of $dr(\mathcal{A})$. Therefore they are pairwise disjoint. From Proposition (4) a right language of $drdr(\mathcal{A})$ is a union of right languages of $rdr(\mathcal{A})$. Since the right languages of $rdr(\mathcal{A})$ are pairwise disjoint, the right languages of $drdr(\mathcal{A})$ are all distinct. Thus, by Proposition (5) the automaton $drdr(\mathcal{A})$ is minimal.

■

5 Analysis of Brzozowski's algorithm

5.1 Split and join for minimizing

Let $\mathcal{A}$ be an automaton which recognizes a regular language L. We study the transformation of the sequence $S_d = (L_d^{\mathcal{A}}(q))_{q\in Q}$ of the right languages of the states of $\mathcal{A}$, when the twofold determinization is performed:

$$S_d \rightarrow_{rdr} S_d^1 \rightarrow_{drdr} S_d^2$$

Notice that since the languages of S_d^1 are pairwise disjoint and the languages of S_d^2 are all distinct, S_d^1 and S_d^2 are sets. Let us remind that the right language of a state is a (left) quotient of L if $\mathcal{A}$ is deterministic and a subset of the intersection of some (left) quotients of L if $\mathcal{A}$ is nondeterministic. The first determinization splits the right languages of $\mathcal{A}$ into disjoint pieces, whereas the second one joins the pieces in order to recombine the set of (left) quotients of L. The effect of the twofold determinization is illustrated by the Example 1. This example is intentionally simple: the initial automaton is deterministic and even minimal.

Example 1

Let q_1 and q_2 be two states of $\mathcal{A}$. We suppose that there exist three distinct words, u, v and w such that: $L_d^{\mathcal{A}}(q_1) = \{u, v\}$, $L_d^{\mathcal{A}}(q_2) = \{v, w\}$, $\{q \mid u \in L_d^{\mathcal{A}}(q)\} = \{q_1\}$, $\{q \mid w \in L_d^{\mathcal{A}}(q)\} = \{q_2\}$ and $\{q \mid v \in L_d^{\mathcal{A}}(q)\} = \{q_1, q_2\}$. We suppose that there exist *two distinct words, s and t such that: $L_g^{\mathcal{A}}(q_1) = \{s\}$, $L_g^{\mathcal{A}}(q_2) = \{t\}$, $\{q \mid s \in L_g^{\mathcal{A}}(q)\} = \{q_1\}$ and $\{q \mid t \in L_g^{\mathcal{A}}(q)\} = \{q_2\}$.*

The determinization of $r(\mathcal{A})$ produces the three states q'_1, q'_2 and q'_3 of $dr(\mathcal{A})$ such that: $q'_1 = \{q_1\}$, $q'_2 = \{q_2\}$ and $q'_3 = \{q_1, q_2\}$. The right languages of q'_1, q'_2 and q'_3 in $rdr(\mathcal{A})$ are pairwise disjoint (they are respectively equal to $\{u\}$, $\{w\}$ and $\{v\}$).

The effect of the first determinization is that the two right languages $\{u, v\}$ and $\{v, w\}$ of $\mathcal{A}$ have been split into three right languages in $rdr(\mathcal{A})$: $\{u\}$, $\{w\}$ and $\{v\}$.

Notice that the left languages of q'_1, q'_2 and q'_3 in $rdr(\mathcal{A})$ are respectively equal to $\{s\}$, $\{t\}$ and $\{s, t\}$ and thus all distinct. This is due to the fact that $\mathcal{A}$ is deterministic (see Proposition (6)).

The determinization of $rdr(\mathcal{A})$ produces the two states q''_1 and q''_2 of $drdr(\mathcal{A})$ such that: $q''_1 = \{q'_1, q'_3\}$ and $q''_2 = \{q'_2, q'_3\}$. The right languages of q''_1 and q''_2 in $drdr(\mathcal{A})$ are distinct (they are respectively equal to $\{u, v\}$ and $\{v, w\}$).

The effect of the second determinization is that the three right languages $\{u\}$, $\{w\}$ and $\{v\}$ of $rdr(\mathcal{A})$ have been joined into two right languages in $drdr(\mathcal{A})$: $\{u, v\}$ and $\{v, w\}$.

5.2 The deterministic case

Brzozowski's algorithm can be applied to a nondeterministic automaton. Here we focus on the case when $\mathcal{A}$ is deterministic. Proposition (6) is due to Brzozowski [3]. Proposition (7) and Corollary (1) are very likely not original. These propositions are gathered in this section for sake of completeness.

Proposition 6 *If $\mathcal{A}$ is deterministic, then $dr(\mathcal{A})$ is the minimal automaton of $r(L)$.*

Proof. Since $\mathcal{A}$ is deterministic, its left languages are pairwise disjoint, and so are the right languages of $r(\mathcal{A})$. The right languages of $dr(\mathcal{A})$, which are unions of right languages of $r(\mathcal{A})$, are therefore all distinct.

■

Proposition 7 *If $\mathcal{A}$ is deterministic, then a state of $rdr(\mathcal{A})$ is a union of Nerode equivalence classes of the automaton $\mathcal{A}$.*

Proof. The transition function of $r(\mathcal{A})$ is denoted by δ_r. Let q_1 and q_2 be two states of $\mathcal{A} = (Q, \Sigma, \delta, i, F)$. We have:

$$q_1 \equiv q_2 \Leftrightarrow [L_d^{\mathcal{A}}(q_1) = L_d^{\mathcal{A}}(q_2) \Leftrightarrow L_g^{r(\mathcal{A})}(q_1) = L_g^{r(\mathcal{A})}(q_2)]$$

Let q' be a state of $dr(\mathcal{A})$. By construction, there exists a word u of Σ^* such that $q' = \delta_r(F, u)$. We have: $q \in \delta_r(F, u) \Leftrightarrow u \in L_g^{r(\mathcal{A})}(q)$. Therefore, q_1 and q_2 are equivalent if and only if they are such that: $q_1 \in \delta_r(F, u) \Leftrightarrow q_2 \in \delta_r(F, u)$. Thus, a state of $rdr(\mathcal{A})$ is a union of equivalence classes of states in $\mathcal{A}$.

■

Corollary 1 *Let $\mathcal{A}$ be a deterministic automaton recognizing a regular language L. Let n be the number of states of $\mathcal{A}$. Let r be the number of (left) quotients of L. Then the deterministic complexity of $r(\mathcal{A})$ is $2^r \leq 2^n$.*

The following proposition leads to a characterization of the equivalence classes of $\mathcal{A}$. It says that two states p and q of $\mathcal{A}$ are equivalent if and only if they belong to the same states of $dr(\mathcal{A})$. This property can be seen as a corollary of Proposition (8).

Proposition 8 *Let p and q be two states of $\mathcal{A}$. It holds:*

$$p \equiv q \Leftrightarrow [p \in P \Leftrightarrow q \in P,\ \forall P \in Q_{dr(\mathcal{A})}]$$

Proof. We have: $p \equiv q \Leftrightarrow [u \in L_d(p) \Leftrightarrow u \in L_d(q), \forall u \in \Sigma^*]$. Moreover $L_d(p) = \bigcup_{p \in P} r(L_g(P))$, with $P \in Q_{dr(\mathcal{A})}$. Hence the result.

∎

6 A variant of Brzozowski's minimization algorithm

We still assume that $\mathcal{A}$ is deterministic. We show that the Proposition (8) leads to an original computation of the equivalence classes of the states of $\mathcal{A}$ after the determinization of $r(\mathcal{A})$ is achieved. On the one hand this result allows us to have a better understanding of how Brzozowski's algorithm performs the minimization: the second determinization actually is a state-equivalence-based procedure. On the other hand it yields a variant of Brzozowski's minimization algorithm, where the second determinization is replaced by a more efficient computation of the equivalence classes.

The Algorithm 1 computes the equivalence classes of $\mathcal{A}$. The partition of Q initially contains two sets: $Q - F$ and F. At each step of the algorithm, a set Y of

Begin
$Partition \leftarrow \{Q - F, F\}$
$Waiting \leftarrow \{F\}$
While $Waiting \neq \emptyset$ **do begin**
  $X \leftarrow First(Waiting)$
  $Waiting \leftarrow Waiting - \{X\}$
  $Processed \leftarrow Processed \cup \{X\}$
  for all $a \in \Sigma$ **do begin**
    $Z \leftarrow \delta_r(X, a)$; **if** $Z \notin Processed$ **then** $Waiting \leftarrow Waiting \cup Z$
  end
  for all $Y \in Partition$ **do begin**
    $K \leftarrow X \cap Y$
    if $K \neq \emptyset$ **then** $Partition \leftarrow Partition \cup K$
    if $X \not\subseteq Y$**then** $Partition \leftarrow Partition \cup (X - K)$
    if $Y \not\subseteq X$**then** $Partition \leftarrow Partition \cup (Y - K)$
  end
end
end

Algorithm 1: Algorithm to extract equivalence classes of $\mathcal{A}$.

the current partition contains possibly equivalent states, in the sense that so far they belong to the same states of $dr(\mathcal{A})$. Every time a new state X of $dr(\mathcal{A})$ is processed, it is checked w.r.t. every set of the partition in order to detect sets containing non-equivalent states of $\mathcal{A}$. The complexity of the Algorithm 1 is exponential since it contains the determinization of $r(\mathcal{A})$. However it is likely more efficient to extract equivalence classes on the fly than performing a second determinization.

7 Conclusion

Brzozowski's minimization algorithm is both simple and mysterious. It is based on two basic and easily understandable operations. However the behaviour of the algorithm is not so obvious. Its average complexity and experimental performance are still unknown or unexplained. This short analysis is intended to contribute to a better understanding of how this algorithm performs the minimization. In particular it shows that the place of Brzozowski's algorithm, in a taxonomy such as Watson's one, is among minimization algorithms based on the computation of a state equivalence.

References

[1] A. Arnold, A. Dicky and M. Nivat, A note about minimal non-deterministic automata, *EATCS Bulletin*, 47, 166–169, 1992.

[2] D. Beauquier, J. Berstel, and P. Chrétienne. *Éléments d'Algorithmique.* Masson, Paris, 1992.

[3] J. A. Brzozowski, Canonical regular expressions and minimal state graphs for definite events, *Mathematical Theory of Automata*, MRI Symposia Series, Polytechnic Press, Polytechnic Institute of Brooklyn, NY,12(1962), 529–561.

[4] C. Carrez, On the Minimalization of Non-deterministic Automaton, *Research Report*, Laboratoire de Calcul de la Faculté des Sciences de l'Université de Lille, 1970.

[5] J.-M. Champarnaud, Subset Construction Complexity for Homogeneous Automata, Position Automata and ZPC-Structures, *Theoret. Comp. Sc.*, 267(2001), 17–34.

[6] F. Coulon, Construction de l'automate canonique d'un langage rationnel, *Mémoire de DEA*, sous la direction de J.-M. Champarnaud, Université de Rouen, 2002.

[7] J. E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Kohavi and Paz, eds, *Theory of Machines and Computation*, Academic Press, New York, 189–196, 1971.

[8] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[9] A. Khorsi, Minimisation des automates finis déterministes, *Master thesis*, sous la direction de D. Ziadi, Université de Sidi-Bel-Abbès, 2002.

[10] S.C. Kleene, Representation of events in nerve nets and finite automata, *Automata Studies, Princeton Univ. Press* (1956) 3–42.

[11] O. Matz and A. Potthoff, Computing Small Nondeterministic Finite Automata, *TACAS'95*, BRICS Note Series NS-95-2, 74–88, 1995.

[12] M. Mohri, Finite-State Transducers in Language and Speech Processing, *Computational Linguistics*, 23:2, 1997.

[13] A. Nerode, Linear Automata Transformation, *Proceedings of AMS*, 9(1958), 541–544.

[14] C. Nicaud, Etude du comportement en moyenne des automates finis et des langages rationnels, Thèse, Université Paris 7, France, 2000.

[15] J.-L. Ponty. Algorithmique et implémentation des automates. Thèse, Université de Rouen, France, 1997.

[16] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res.*, 3(2):115–125, 1959.

[17] D. Revuz, Minimization of Acyclic Deterministic Automata in Linear Time, *Theoret. Comput. Sci.* 92(1992), 181–189.

[18] B. Watson, Taxonomies and Toolkits of Regular Languages Algorithms, PhD thesis, Eindhoven University of Technology, The Nederlands, 1995.

[19] B. Watson, Directly Constructing Minimal DFAs: Combining Two Algorithms by Brzozowski, in S. Yu and A. Paun, eds, CIAA 2000, London, Ontario, *Lecture Notes in Computer Science*, 2088(2001), 311–317, Springer.

[20] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume I, Words, Languages, Grammars, pages 41–110. Springer-Verlag, Berlin, 1997.