Collaborative Report DC–2001–06

Proceedings of the Prague Stringology Conference '01

Edited by Miroslva Balík and Milan Šimánek

September 2001

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, Bruce W. Watson

Organizing Committee

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

Table of contents

Preface

This collaborative report contains the proceedings of the Prague Stringology Conference '01 (PSC'01), held at the Department of Computer Science and Engineering of Czech Technical University in Prague on September 4, 2001. The conference was preceded by workshop PSCW'96 which was the first action of the Prague Stringology Club, and continued each year. The proceedings of PSCW'96, PSCW'97, PSCW'98, PSCW'99 and PSCW 2000 were published as collaborative reports DC–96–10, DC–97–03, DC–98–06, DC–99-05 and DC-2000-03, respectively, of Department of Computer Science and Engineering and are also available in the postscript form at Web site with URL: `http://cs.felk.cvut.cz/psc`. While the papers of PSCW'96 were invited papers, since 1997 they were selected from the papers submitted as a response to a call for papers. The papers in this proceedings are alphabetically ordered by the authors. The last two papers are invited.

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

Miroslav Balík and Milan Šimánek, editors

Searching in an Efficiently Stored DNA Text Using a Hardware Solution

T. Berry, S. Keller and S. Ravindran

Department of Computer Science
Liverpool John Moores University
Byrom Street
Liverpool
United Kingdom

e-mail: {T.Berry, S.Keller, S.Ravindran}@livjm.ac.uk

Abstract. In this paper, we describe a storage method that reduces the size of a DNA text file to 25% of its original size. Also outlined is a new algorithm, which can search an input stream of DNA text for multiple DNA sub-strings in a single pass. Although this new algorithm is competitive when compared to the majority of existing string matching algorithms, the intention is to further improve performance by implementing the algorithm as a hardware-only solution.

1 Introduction

String matching and Compression are two widely studied areas in computer Science [10]. String matching is detecting a pattern P of length m in a larger text T of length n. Compression involves transforming a string into a new string which contains the same information but whose length is as small as possible. These two areas naturally lead to Compressed String Matching, i.e. searching for a pattern in a compressed text. This method will save both space and time.

In this paper we describe a hardware solution that searches in the compressed DNA text. We also describe an algorithm coded in the programming language C that will be synthesized into hardware. A DNA text (or molecule) encodes information which by convention is represented as a string over the DNA alphabet A, C, G and T. Compressed String Matching in a DNA text is useful for the following reasons. Although the cost of memory is reducing, the sizes of DNA databases are growing exponentially.

Optimal compression will devote two bits to represent each DNA character, if each character is drawn uniformly at random from the DNA alphabet and that all positions in the text are independent [14]. The compression method described in Section 2 also devotes two bits per character, i.e. the method guarantees to compress the DNA text to 25% of its original size. Section 3, outlines the BK algorithm, as being a string matching algorithm, which as well as being relatively fast as a software solution, could also be implement in a hardware-only solution. Section 4, describes a modification to the basic BK algorithm, which will search a stream of

DNA text for multiple sub-strings in a single pass of the text. Section 5, covers the process of implementing programs as hardware-only solutions. Attention is paid to the inadequacies of modern microprocessors and the advantages which so-called 'hardware compilation techniques' can offer as a means of accelerating the execution of algorithms. Section 6, describes how the BK string matching algorithm may be implemented as a hardware only solution. In section 7 we describe 5 existing string matching algorithms. In section 8 we compare our new algorithm with the 5 existing algorithms by experimentation. The texts and the patterns used for these experiments have been taken from [11] and [1] respectively.

2 Efficient storage of a DNA text

In the DNA alphabet, S, there are four characters, namely A, C, G and T. As there are only 4 possible characters in a DNA text we can represent the character's with the function, f: $\Sigma \Rightarrow [0 .. 3]$, such that:

$$f(A) = 0, f\ (C) = 1, f\ (G) = 2, \text{ and } f\ (T) = 3.$$

Let a block be a string of four characters. The code of a block of DNA characters is the value that returned by the function g, g: $\Sigma \times \Sigma \times \Sigma \times \Sigma \Rightarrow [0 .. 255]$, for the block. The function g is defined as follows. $g(\alpha\beta\gamma\delta) = (f(\alpha) \times 4^3) + (f(\beta) \times 4^2) + (f(\gamma) \times 4^1) + (f(\delta) \times 4^0)$

This means that we can represent each of the DNA characters with 2 bits. Namely A = 00, C = 01, G = 10 and T = 11. A DNA text block will be represented by 32-bits, as each character needs 8-bits. Using the function g we can represent a text block with 8-bits. For each text block we print an ASCII character whose ASCII number is the value return by the function g. As the function g is a bijective function, we can compress any text block into 8-bits and it is possible to reconstruct the original DNA text exactly.

For example, CAAGAGCGCAGT $\Rightarrow$ 010000100010011001001011 $\Rightarrow$ 66 38 75 $\Rightarrow$ B&K. So we can store the string CAAGAGCGCAGT using 24 bits. This storage method will guarantee to store the DNA text in a file, which is 25% of its original size.

3 Investigation into a hardware only solution to the string matching problem

The string matching algorithm illustrated in Figure 1 was devised as part of a case study to investigate the feasibility of performing computational algorithms in hardware. String matching was chosen as one of the areas to be tested as such algorithms typically involve many hardware manipulations of words of binary data. These manipulations are invoked by the machine code instructions, which constitute the program and performed by the general-purpose hardware within the microprocessor itself. So called software to hardware synthesis techniques aim to accelerate algorithm execution by first of all removing the need for machine instructions and by also performing computational and logical operations on bespoke hardware.

```
while (match != 0 && word_count != 0) {
      result = current & mask;
      match  = result - target;
      if (match != 0) {
         current = current >> 2;
         temp = buffer << 14;
         current = current | temp;
         if (shifted == 7) {
            word_count--;
            shifted = 0;
            buffer = *ptr;
            ptr++;
         }
         else {
            buffer = buffer >> 2;
            shifted++;
         }
      }
   }
```

Figure 1: The C code for searching for occurrences of a single pattern in a given text

The example code shown works on a word size of 16 bits and can detect a pattern of up to 8 DNA characters in length. However, the algorithm is by no means limited to this word size.

The algorithm works by shifting the input stream through the variable *current*. When the data is shifted, it is shifted two bits at a time to the right. It is shifted two bits at a time because this is more efficient as the algorithm are searching for DNA features which are encoded into two bit patterns. Each time *current* is shifted to the right it is checked for a match with the target pattern. This concept is illustrated in Figure 2.

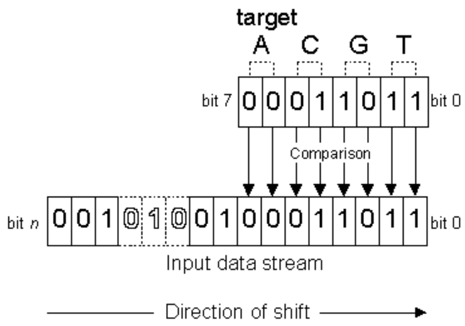

Figure 2: Comparison of input stream against target

When shifted, the two least significant bits (LSBs), which are bits 1 and 0, are lost and the two most significant bits (MSBs), which are bits 15 and 14, become empty. These two null MSBs are filled with the two LSBs of $buffer$. The variable $buffer$ is a pre-fetch word, which will contain word i+1 with *current* containing word i. This is necessary if *current* is to kept full at all times. During initialisation, the first word of data is copied to *current* from the input buffer and $buffer$ is filled with the second word of data.

In order to copy the two LSBs of $buffer$ to the two MSBs of *current*, $buffer$ is first copied to a variable *temp*, which is then shifted 14 bits to the left. This shift operation results in the two least significant bits of buffer (1 and 0) being moved to the two most significant bits (15 and 14), with the remainder of the word (bits 13 to 0) being filled with 0's. The contents of *temp* is then ORed with *current* resulting in the two most significant bits of *current* being replaced with the two least significant bits of *buffer*.

In order to make sure that *buffer* always has at least two bits available for *current*, a count is kept of how many times *current* has been shifted to the right. This count is stored in the variable *shifted*, which is initialised to 0 and then incremented each time *shifted* is shifted to the right and the two MSBs replaced with the two LSBs of *buffer*. If after a comparison *shifted* is less than 7, then *buffer* is shifted two bits to the right in order to replace the two LSBs which have been moved to *current* and the variable *shifted* is incremented. If shifted reaches 7, then the last two bits of data have moved from *buffer* to *current* and *buffer* requires re-filling. When this occurs, *shifted* is set back to 0 and *buffer* is loaded with a complete new word from the input stream.

The next byte to be fetched from the input stream is pointed to by the pointer variable *ptr*, which is incremented once *buffer* has been refilled with a word from data buffer named *data_buffer*.

To ascertain whether *current* contains a match for the bit pattern being searched for, *current* is first ANDed with a variable named *mask*. The purpose of *mask* is to mask out those bits of *current* which are not required for the comparison. To ignore a bit during the comparison between *target* and *current*, then the associated bit of *mask* should be 0. Likewise, to include a bit in the comparison, then that bit of the mask should be set to 1. As illustrated in Figure 3 below, the pattern 'ACGT' is being searched for, which is only an 8 bit pattern. Hence the remaining upper eight bits can be ignored during the comparison and are thus set to 0.

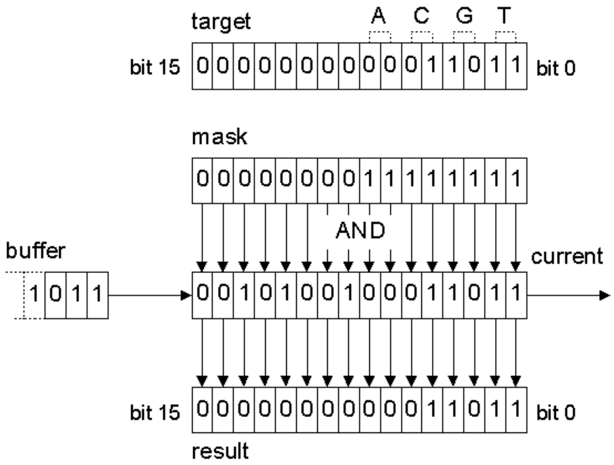

Figure 3: The use of the mask to reduce the number of bits compared

When *current* is ANDed with the mask, the result of the logical AND is stored in *result*. A bit of *result* will only be set to 1 if both the corresponding bits of *mask* and *current* are 1, otherwise the bit will be set to 0. A match with the target can now be determined by subtracting *target* from *result*. If the result of this subtraction is all 0's, then both result and the target must have contained the same values and hence a match has been found. This process is illustrated in Figure 4.

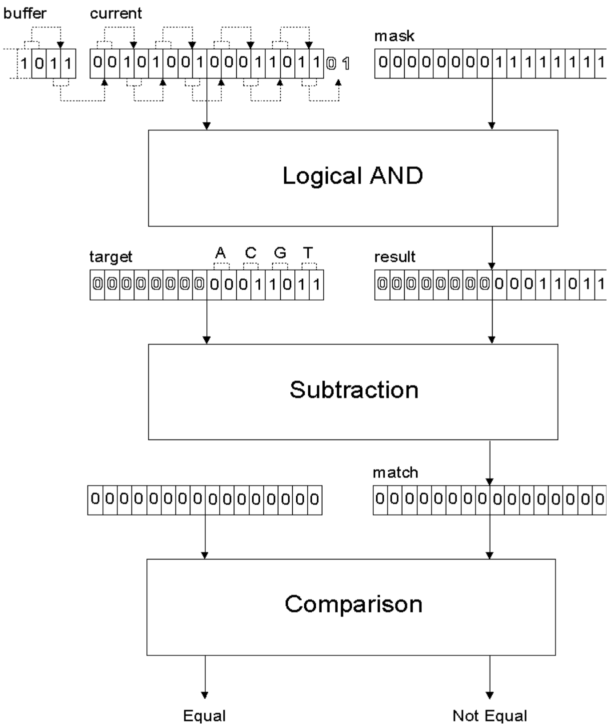

Figure 4: The steps required to determine whether the target matches the current data

The program has been written to locate patterns of DNA up to and including eight two bit codes. Hence, all words are 16bits in length and are declared as being of type *unsignedshort*. However, the program could easily be amended to locate longer patterns by simply changing the variable types and program constants.

4 Searching for multiple strings

The example algorithm illustrated in Figure 1, simply searches an input stream for all occurrences of a single string. The program can be easily modified to search an input stream for all occurrences of many strings by reading in many targets from a file and storing them in an array. This way, each time *current* is shifted, it may be compared with many targets before it is once more shifted. In order to do this, a second array must be created to store the masks for each of the targets. These masks may be automatically generated from the targets as they are read in.

```
while (shifts>0) {
      for (i=0; i<no_of_targets; i++) {
         result = current & mask_array[i];
         match  = result - target_array[i];
         if (match == 0) {
          .. match found
         }
      }

      current = current >> 2;
      shifts--;
      temp = buffer << 14;
      current = current | temp;

      if (shifted == 7) {
         shifted = 0;
         buffer = *ptr;
         ptr++;
      }
      else {
         buff = buff >> 2;
         shifted++;
      }
   }
```

Figure 5: An algorithm to search for multiple patterns in a single text

Apart from this simple modification, the program remains relatively unchanged. This is the version of the program, which will be the subject of the investigation into hardware acceleration of string matching.

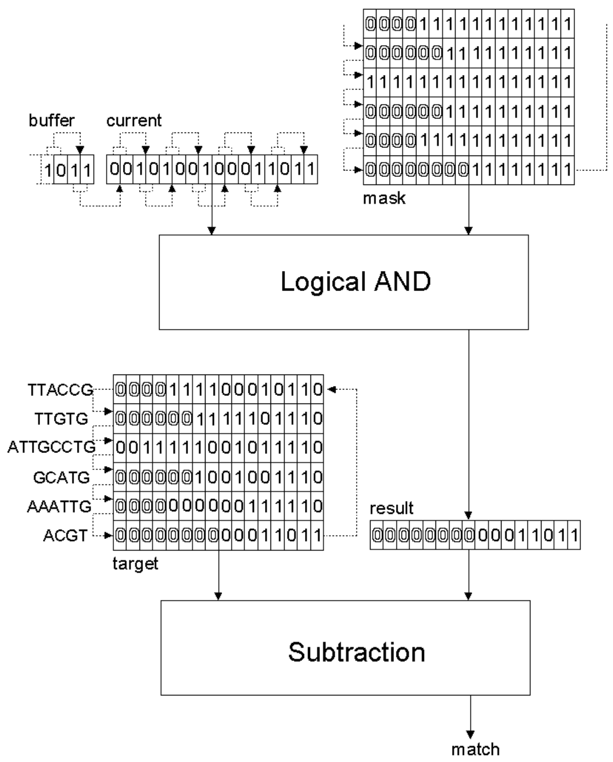

Figure 6: Illustration of Figure 5

5 Hardware acceleration

Over the past decade hardware synthesis has been explored as a method of accelerating computing tasks at which conventional general-purpose microprocessors are inefficient. The problem is that current microprocessors, although being suitable for many tasks, are not particularly efficient at performing any one task. This is because they are designed to be applicable to as many problem areas or tasks as possible. Therefore, through necessity they possess many features which although utilised by one application may never be used by another application. Another problem with conventional machines is the stored program concept whereby and algorithm is executed by the microprocessor obeying a series of commands, which are stored in memory. These commands are the machine code instructions, which the microprocessor fetches, decodes and then executes one at time. This fetching and decoding takes comparatively vast amount of time due to the slow speed of memory and the numerous instructions within the instruction set of the processor. Even the execution phase is by no means efficient. The execution circuits of a processor are finite and although some resources are replicated, many must be shared. This resource contention slows execution times. Additionally, the execution circuits of microprocessors are designed to perform many tasks, making them less efficient.

Hardware and software co-design or hardware to software synthesis is a process whereby computing algorithms expressed in high-level languages, are compiled to produce either an executable program and a hardware circuit design or just a hardware circuit. In the case of hardware and software co-design [16, 17], the majority of the program is turned into an executable binary for execution on a microprocessor, whilst the remainder of the algorithm is synthesised to hardware. The portion synthesised to hardware would be the section of the algorithm at which the microprocessor would be least efficient. The hardware portion is usually programmed into a Field Programmable Gate Array (FPGA) [18], which then acts as a co-processor to the host microprocessor. Producing programs for such architectures is usually performed using a hybrid programming language and appropriate compilers and synthesis tools [15]. Such programming languages tend to be based on C, with extensions being added to express the hardware-only components for the FPGA.

With pure software to hardware synthesis [2, 3], an attempt is made to map the entire algorithm into an FPGA, resulting in a digital circuit, which is functionally identical to and directly derived from an algorithm, which was originally expressed in a programming language. Such approaches tend to used hardware description languages such as VHDL [13], which are exclusively used for expressing the function of hardware circuits.

Synthesis to a hardware only solution offers the greatest potential increase in speed, removing the need for instructions and a conventional fetch-decode-execute cycle. However, it is also the most difficult to achieve. The difficulty arises from the design features of current FPGAs, which were originally intended for implementing digital circuits. Although suitable for the prototyping and implementation of general circuits comprising of digital logic, they are not well suited for implementing algorithms. This is because algorithms require data storage for variables, buses for register to register and register to execution unit transfers. Data storage and buses are not available within an FPGA and must be created using the FPGAs resources, such as macro-cells and signal lines. What makes the situation worse is that both registers and buses are expensive in terms of FPGA resources, which ultimately limits the size of the algorithm to may be implemented in hardware.

As part of the research into implementing string matching algorithms in hardware-solutions, recommendations will be made regarding the development of a new FPGA architecture, which will be more suited to purpose of implementing software in hardware.

6 Hardware Implementation of string Matching

The research currently being undertaken aims to overcome the limitations of current FPGAs, with regards to configurable computing. First of all, it aims to do this by recommending a new configurable device architecture, which lends itself more to the mapping of software to hardware. The device will feature the busing systems, areas of storage and synchronization circuits required to facilitate both effective and efficient hardware generation. Secondly, software tools are being developed which will process standard C programs and as their output, will produce configuration files for the programmable device.

Because of the low-level nature of the task of string matching, it is an ideal candidate for such acceleration techniques. At the hardware level, the most efficient method of searching a string for a sub-string is as illustrated in Figure 2. The stream to be searched is passed through a register, shifting one bit at a time. Each time the register is shifted, the register is compared with the sub-string being searched for. This is the same method as employed in the C algorithm discussed previously. The number of register bits to be compared need only be equal in length to the number of bits in the sub-string, with any additional bits simply be masked out or ignored in the same way as the C algorithm. Additionally, the register being searched need not only be shifted one bit at a time. In the case of searching for occurrences of bit patterns consisting of two bit sub-patterns, it is more efficient to shift the register two bits before a comparison with the target is made.

Figure 7., illustrates a simplified diagram of the components to be implemented in hardware. Missing are the hardware components responsible for shifting both $current$ and $buffer$ to the right. Also missing are the circuits required for synchronizing the activities of the components in order to perform the operations of the algorithm in the correct order.

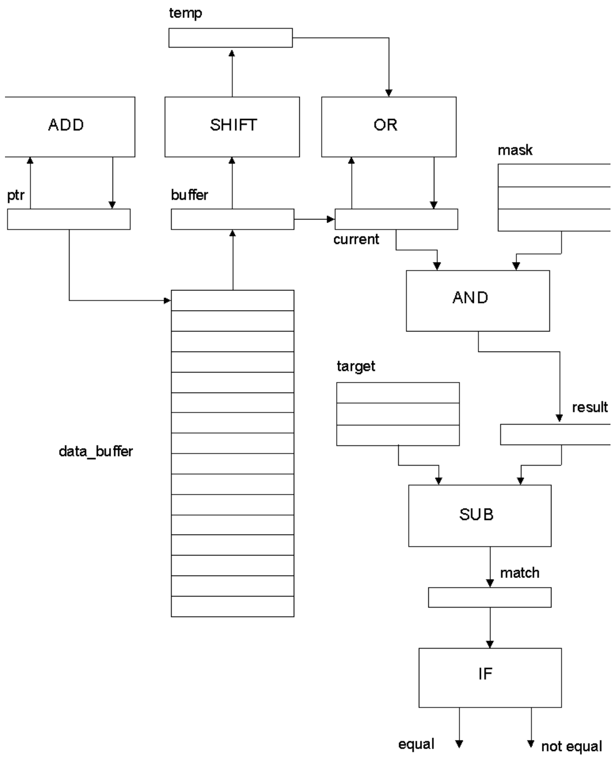

Figure 7: Simplified version of the components to be implemented in hardware

The memory labelled *data_buffer* holds the data to be searched for a sub-string. The width of the words contained in *data_buffer* is immaterial and may be of any width.

The registers labelled *ptr* and *buffer* are associated with the fetching of the words from memory. The register *buffer* is the same width as the words contained in *data_buffer*. This register is used to contain a pre-fetch word. The register *ptr* is used as a pointer to reference the words contained in *data_buffer*. As such, its width need only be sufficient to reference all of the words in *data_buffer*.

The register *current* contains the current bit pattern to be matched against a sub-string bit pattern. It is a shift register, with the data contained in the register being shifted right two bits at a time, with the two least significant bits being lost and the two most significant bits being replaced with the two least significant bits of buffer. This is the purpose of buffer, to keep current full of bits. Only once all the bits contained in buffer have been shifted into current, will new data be loaded into *buffer* from *data_buffer*.

As with the C algorithm, the mask register is used to contain a bit pattern to mask off the bits of *current*, which are not to be compared. When ANDed with the contents of *current*, then the resulting word is stored in the register *result*. It is the contents of *result*, which will be compared with the target to determine whether or not a matching bit pattern has been located. To ascertain whether the contents of result and target do match, result is subtracted from target. Again, if the result of the subtraction is zero, then a match has been located.

The synchronisation techniques to be implemented to synchronise the functioning of the component parts is beyond the scope of this paper. However, the techniques employed and the architecture of the programmable logic device, will be reported upon in subsequent papers.

7 Existing string matching algorithms

The string matching algorithms described below work as follows. First the pattern of length m, P[1..m], is aligned with the extreme left of the text of length n, T[1..n]. Then the pattern characters are compared with the text characters. The algorithms vary in the order in which the comparisons are made. After a mismatch is found the pattern is shifted to the right and the distance the pattern can be shifted is determined by the algorithm that is being used. It is this shifting procedure, which is the main difference between the string matching algorithms.

There are a number of string matching algorithms available in the literature. We have chosen six of them, which were found to be fast in [5] and described them briefly below. All of the algorithms have worst-case search time O(nm). Animations of these algorithms can be found at [9] and more information about the algorithms can be found in [8].

In the Boyer-Moore (BM) algorithm [7] the characters are compared from right to left starting with the rightmost character of the pattern. In a case of mismatch it uses two functions, last occurrence function and good suffix function and shifts the pattern by the maximum number of positions computed by these functions. The good suffix function returns the number of positions for moving the pattern to the right by the least amount, so as to align the already matched characters with the rightmost substring in the pattern that are identical. The number of positions returned by the last occurrence function determines the rightmost occurrence of the mismatched text character in the pattern. If the text character does not appear in the pattern then the last occurrence function returns m.

The Horspool (HOR) algorithm [12] is a simplification of the BM algorithm. It does not use the good suffix function, but uses a modified version of the last occurrence function. The modified last occurrence function determines the right most occurrence of the $(k+m)^{th}$ text character, T$[k+m]$ in the pattern, if a mismatch occurs when a pattern is aligned with T$[k\ ..\ k+m]$. This algorithm changes the order in which characters of the pattern are compared with the text. It compares the rightmost character in the pattern first then compares the leftmost character, then all the other characters in ascending order from the second position to the $m-1^{th}$ position.

The Berry-Ravindran (BR) algorithm [5] uses the next two characters outside the pattern text alignment T$[k+m+1]$ T$[k+m+2]$ to calculate the shift. If the pair is in the pattern then we shift the pattern so as to align it with the rightmost occurrence of the pair in the pattern. If the pair is not in the pattern then we shift by m+2 places to the right.

The DS algorithm [6] is an algorithm designed to search directly in the efficiently stored DNA text. It was found to be the fastest algorithm for the task of string matching in DNA files. The speed of the algorithm was mainly due to the cut down in the time required to scan in the text due to it being 25% of the size of the original text. The DS algorithm has a worst case run time of O(nm) but an average case run time of O($n+m$). The algorithm compares text blocks with pattern blocks directly to see if they match. Upon a mismatch the algorithm moves to the next text block to be considered.

The Shift-OR (SO) algorithm [4] has a worst case run time of O(n) independent of the size of the alphabet being used or the pattern being searched for. The SO algorithm constructs a bit array R of length m. The array has the initial state R_i and $R_i[j]$ is equal to 0 if P[0,j] = T[$i - j$,i] for all $0 \leq j \leq m$. Otherwise $R_i[j]$ is equal to 1. R_i is recalculated to form $R_i + 1$ by using two operations a logical shift of 1 and a logical OR hence the name of the algorithm Shift-OR.

8 Comparison with existing algorithms

We measure the user time for the six algorithms. We timed the search of each of the 5 texts randomly chosen from the Entrez database [11] for all occurrences of the 62 enzyme cutting boundaries in [1]. There are 9 patterns of length 4, 50 of length 6 and 3 of length 8. The BK and DS algorithms searched in the efficiently stored DNA text file and the BM, HOR, BR and SO algorithms searched in the original DNA text file. We used a 486-DX66 with 32 megabytes of RAM and a 100 megabyte hard drive running SUSE 5.2. The user time includes the time taken for any pre-processing and the reading of the text into memory. Each algorithm was evaluated ten times and the average user time taken is given in Table 1. The timings were accurate to 1/100 of a second. The difference between the slowest and fastest time for each test for an algorithm was less than 0.1 of a second.

Text	Text size	BM	BR	HOR	SO	DS	BK
1	100,000	47.57	31.49	41.09	54.92	15.31	45.77
2	100,000	47.63	31.46	40.99	54.91	15.36	45.76
3	253,505	119.97	79.29	102.97	138.96	33.18	115.92
4	319,000	151.13	99.63	129.70	174.71	40.84	145.83
5	217,000	102.72	67.98	88.26	118.85	29.05	99.22

Table 1: The user time taken (given in seconds) to search for all 62 patterns in each of the texts

¿From Table 1 we can see that the DS algorithm is the fastest algorithm for the task. This is due to the savings made by the algorithm searching in the compressed DNA file, which is a quarter of the size of the original DNA text file. The BR algorithm is the best algorithm for searching in the original DNA text file. This is due to the larger shift of m+2 given by this algorithm. Using two characters to perform the search means that the probability of a large shift is increased. We would expect the average shift for the algorithm to be greater than m for all the patterns searched for. The BK algorithm is faster than the BM and the SO algorithms. The BK algorithm is a C implementation of our proposed hardware solution. We expect our hardware solution to be faster than our C implementation, which will also be faster than the DS algorithm.

9 Conclusion

Using the storage method described in Section 2 we can store DNA text files in 25% of space required for the original DNA text file. Using algorithms such as the DS and BK algorithm we can keep DNA texts efficiently stored and perform searches on them. Thus saving both time and space.

Although the BK algorithm, which is presented in this paper, is not the fastest algorithm for the task of string matching in an efficiently stored DNA text file, it is never-the-less still competitive when compared to existing string matching algorithms. Although it is by no means the fastest algorithm for sub-string searches, the hardware synthesis of the BK algorithm into a hardware only implementation is expected to produce a solution that we estimate to be significantly faster than even the DS algorithm.

References

References

[1] Amersham life science products catalogue, pp 378-379, 1998.

[2] James B. Peterson, R. Brendan O'Connor, Peter M. Athanas, *"Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures"*, The Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

[3] James B. Peterson, Peter M. Athanas, *"High-Speed 2-D Convolution with a Custom Computing Machine"*, The Bradley Department of Electrical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

[4] Baeza-Yates R. A., Gonnet G. H., *"A New Approach to Text Searching"*, Communications of the ACM, 35(10), pp. 74-82, 1992

[5] Berry T., Ravindran S., *"A fast string matching algorithm and experimental results"*, Prague Stringology Club Workshop '99, 1999.

[6] Berry T., Ravindran S., *"String matching in a compressed DNA text"*, Proceedings of the Australian Workshop on Combinatorial Algorithms (AWOCA '99), pp. 53-62, 1999.

[7] Boyer R. S., Moore J. S., *"A fast string searching algorithm"*, Communications of the ACM, 23(5), pp 1075-1091, 1977.

[8] Charras C., Lecroq T., 1997, Exact string matching, available at: `http://www-igm.univ-mlv.fr/~lecroq/string.ps`

[9] Charras C., Lecroq T., 1998, Exact string matching animation in JAVA available at: `http://www-igm.univ-mlv.fr/~lecroq/string/`

[10] Crochemore M., Rytter W., *"Text algorithms"*, Oxford University Press, 1994.

[11] Entrez database available at: http://www.ncbi.nlm.nih.gov/Entrez/

[12] Horspool R. N., *"Practical fast searching in strings"*, Software Practice and Experience, 10(6), pp 501-506, 1980.

[13] *"IEEE Standard VHDL Language Reference Manual"*, The Institute of Electrical and Electronics Engineers, Inc. (1987)

[14] Loewenstern D., Yianilos P., *"Significantly lower entropy estimates for natural DNA sequences"*, Journal of Computational Biology, 6(1), 1999.

[15] Page I., Luk W., *"Compiling occam into FPGAs"*, in FPGA, eds., Will Moore and Wayne Luk, 271-283, Abingdon EE&CS books, (1991).

[16] Page I., *"Constructing Hardware-Software Systems from a Single Description"*, Oxford University Computing Laboratory.

[17] Page I., Aubury M., Randall G., Saul J., Watts R., *"hcc: A Handel-C Compiler"*, Oxford University Computing Laboratory.

[18] Xilinx Inc., *"Spartan and SpartanXL Families of Field Programmable Gate Arrays"*, Preliminary Product Specification". San Jose, CA, (1999)

A linear time string matching algorithm on average with efficient text storage

T. Berry and S. Ravindran

Department of Computer Science
Liverpool John Moores University
Byrom Street
Liverpool
United Kingdom

e-mail: T.Berry@livjm.ac.uk,S.Ravindran@livjm.ac.uk

Abstract. In this paper we describe an algorithm to search for a pattern in an efficiently stored text. The method used to store the text tranforms it to $\frac{\lceil \log_2 \sigma \rceil}{8}$ of its original size, where σ is the size of the alphabet set Σ. We prove that the algorithm takes linear time on average. We compare the new algorithm with some existing string matching algorithms by experimentation.

1 Introduction

String matching and Compression are two widely studied areas in computer science [5]. String matching is detecting a pattern P of length m in a larger text T of length n over an alphabet set Σ of size σ. Compression involves transforming a string into a new string which contains the same information but whose length is as small as possible. These two areas naturally lead to compressed string matching, i.e. searching for a pattern in a compressed text. This method will save both space and time.

In this paper we describe a string matching algorithm to search a pattern in a efficiently stored text. We can reduce the size of any given text according to the size of the alphabet being used. This is useful as although the cost of memory is reducing, the sizes of text databases are growing exponentially.

In Section 2 we describe our storage method that will transform the text to $\frac{\lceil \log_2 \sigma \rceil}{8}$ of its original size. This method is compared with other well-known compression algorithms by experiments in Section 3.

Section 4 describes a novel string matching algorithm on a text that is stored by using the method in Section 2. In Section 5 we prove that on average this algorithm takes $O(n + m)$ time. Our algorithm is compared with other well known string matching algorithms by experiments in Section 6.

2 Efficient storage of a text

We assume that the size of the alphabet set, σ, is in the range $1 \leq \sigma \leq 128$ and that we are representing each character in the alphabet with one byte. There are redundant bits in each byte as we only need $\lceil \log_2 \sigma \rceil$ bits to represent a character.

After we replace the characters in a text with $\lceil \log_2 \sigma \rceil$ bits, it is possible to replace eight consecutive bits in the binary text with its corresponding ASCII character. These eight consecutive bits are called a *block*. The decimal value of a block is the *code* of the block. This representation will reduce the storage space to $\frac{\lceil \log_2 \sigma \rceil}{8} n$, where n is the size of the original text.

For example, consider $\Sigma = \{A, B, C, D, E\}$ and $T = CACDABEB$. This text T of eight characters can be represented with three characters $T' = A1a$. First we represent the characters with $A = 000, B = 001, C = 010, D = 011$ and $E = 100$. This will give the binary representation of the text T:

01000001**0**0110000**0**1100001

The first bit in each block are shown in bold font. The codes for the text blocks are 65, 48 and 95 and their corresponding ASCII characters are 'A', '1', and 'a' respectively.

3 Comparison with existing algorithms

The method described in the last section is not compression as in the literature but does reduce the size of the original text. In this section we compare the well known text compression methods, Huffman encoding [8] and Lempel-Ziv encoding [9, 13, 14] with our method.

The Huffman encoding determines the length of the bit representation of the characters according to their frequency. It assigns smaller codes to high frequency characters and larger codes to low frequency characters.

In Lempel-Ziv (LZ) encoding [14] the file may be compressed to less than $\lceil \log_2 \sigma \rceil$ bits per character but requires re-occurring strings. Each of the repeated strings and each of the characters in the alphabet are represent by 12 bits. The gains from this method are reliant on there being enough repeated strings to counter the 12 bits which are used to represent each of the compressed strings.

The LZ encoding and its derivative LZW encoding [13] are used in UNIX utilities, compress and gzip. Another variation of LZ encoding (NR) is described in [9].

Table 1 (see Appendix) shows that our storage method is comparable to these methods. Although our method is not very good for text files with large alphabets. The method is competive for DNA, Recombinant DNA (RDNA) and hexadecimal files. Note that the main purpose of this paper is not compression, but for the searching of a pattern in a compressed file.

4 Searching in a text with efficient storage

In this section we describe an algorithm to find all exact occurrences of a pattern in a text. Here we assume that the text is stored as described in Section 2 and $\sigma \leq 128$. We describe the algorithm for $\sigma = 2$, we will see later that the algorithm can be easily adapted for $\sigma > 2$.

A substring of the pattern may overlap between consecutive text-blocks and a pattern may start in a text-block at any one of eight positions. During the search we need to look whether a prefix (or suffix) of a pattern is a suffix (or prefix) of a text-block. Due to this problem we have to consider eight different expressions. Each expression is made up of pattern-blocks of length eight bits. There will be $m + 7$ pattern-blocks, where m is the length of a pattern.

For a pattern $\mathrm{P}_1\mathrm{P}_2..\ \mathrm{P}_m$ we can construct the expressions as shown in Figure 1. Here we consider the case for m mod $8 = 0$. We number the pattern-blocks starting from 0 at the top left corner to $m+6$ in the bottom right corner as shown in brackets. The wildcard character N represents either 0 or 1, and $\mathrm{P}_{i..j}$ represents $\mathrm{P}_i..\mathrm{P}_{j-1}\mathrm{P}_j$, for $1 \leq i < j \leq m$.

Exp0:	NNNNNNNP_1	(0)	....	$\mathrm{P}_{m-14..m-7}$	(m-8)	$\mathrm{P}_{m-6..m}$N	(m)
Exp1:	NNNNNN$\mathrm{P}_{1..2}$	(1)	....	$\mathrm{P}_{m-13..m-6}$	(m-7)	$\mathrm{P}_{m-5..m}$NN	$(m+1)$
Exp2:	NNNNN$\mathrm{P}_{1..3}$	(2)	....	$\mathrm{P}_{m-12..m-5}$	(m-6)	$\mathrm{P}_{m-4..m}$NNN	$(m+2)$
Exp3:	NNNN$\mathrm{P}_{1..4}$	(3)	....	$\mathrm{P}_{m-11..m-4}$	(m-5)	$\mathrm{P}_{m-3..m}$NNNN	$(m+3)$
Exp4:	NNN$\mathrm{P}_{1..5}$	(4)	....	$\mathrm{P}_{m-10..m-3}$	(m-4)	$\mathrm{P}_{m-2..m}$NNNNN	$(m+4)$
Exp5:	NN$\mathrm{P}_{1..6}$	(5)	....	$\mathrm{P}_{m-9..m-2}$	(m-3)	$\mathrm{P}_{m-1..m}$NNNNNN	$(m+5)$
Exp6:	N$\mathrm{P}_{1..7}$	(6)	....	$\mathrm{P}_{m-8..m-1}$	(m-2)	P_mNNNNNNN	$(m+6)$
Exp7:	$\mathrm{P}_{1..8}$	(7)	....	$\mathrm{P}_{m-7..m}$	(m-1)		

Figure 1: Expressions for a pattern $\mathrm{P}_1\mathrm{P}_2..\ \mathrm{P}_m$ when m mod $8 = 0$.

The naive algorithm will compare a text-block with the first pattern-blocks in each expression. If any of these pattern-blocks matched with the text-block, we need to compare the consecutive text-blocks with the rest of the pattern-blocks in the expression.

Our algorithm first constructs a table called the *Block-Table*. The Block-Table has 256 columns and $m + 7$ rows as there are 256 possible blocks in a text and m+7 is the number of pattern-blocks we need to consider. The table is initialised to 0. The $(i, j)^{th}$ entry in the table is defined as follows, where i, $0 \leq i \leq m + 6$, is the pattern-block number and j, $0 \leq j \leq 255$, is the code for a block. Suppose that the pattern-block does not have a wildcard character, the $(i, j)^{th}$ entry is 1, if the code for pattern-block i is equal to j. If there is one or more wild cards in the pattern-block, we consider all the possible blocks. For example, if the i^{th} pattern-block is NN111000, the $(i, j)^{th}$ entry is equal to 1 for all j, where j is the code for 00111000, 01111000, 10111000 or 11111000.

For each expression we only have to compare one pattern-block with a text-block, and if these two match then we compare the rest of the pattern-blocks in the expression with the corresponding text-blocks. We choose a pattern-block (from each expression) which has the minimum number of possibilities of matching with a text-block. We build the *Order-Table* of dimensions 8 by $\lceil \frac{m+7}{8} \rceil$ which contains the order in which to examine the pattern-blocks for each expression. For each pattern-block the number of possibilities of matching a text-block can be found by adding the values in the row of the pattern-block in the Block-Table.

¿From these we construct a *Search-Table* of dimensions 8×256, and it is initialised to -1. In the first row of the Search-Table, we enter pattern-block numbers from the first column of the Order-Table. If j is the code for these pattern-blocks, we enter the pattern-block numbers at the j^{th} column, for all j, $0 \leq j \leq 255$. A column number may be the code for more than one of the chosen pattern-blocks. This is because a text-block can match pattern-blocks from more than one expression. As there are only eight expressions we need a maximum of eight rows. For example, the chosen pattern-blocks, 110011NN and NN001100, will both match the block 11001100. We enter the pattern-blocks (110011NN and NN001100) numbers in the first and second rows respectively of the column k, where k is the code for 11001100.

We begin the search at the begining of the text and compare the text-blocks with chosen pattern-blocks in the Search-Table. We check the j^{th} column in the Search-Table, where j is the code of the text-block. If the entry is -1 then we check the next text-block. Otherwise we know that the text-block is in the pattern. We compare the rest of the pattern-blocks in the expression with the corresponding text-blocks until either full match or mismatch is found using the Block-Table and Order-Table. Before we move to the next text-block, we check if the entry in the next row of the Search-Table is -1. We repeat this process if the entry is not -1, otherwise we check the next text-block.

If $\sigma > 2$, we have to convert the pattern into a binary string by mapping the characters into $\lceil \log_2 \sigma \rceil$ bits as we did in Section 2. Here we don't have to consider all the expressions. This is because in the pattern-blocks 0, 1, .. , 7 (from expressions 0 to 7 respectively) the pattern starts at positions 7, 6, .. , 0 respectively (see Figure 1). The positions are numbered from left to right in a pattern-block.

We can show that for all σ, in a comparison we need at most $\lceil \frac{8}{\lceil \log_2 \sigma \rceil} \rceil$ expressions. There are two cases which depend on whether $\frac{8}{\lceil \log_2 \sigma \rceil}$ is an integer.

Suppose $\frac{8}{\lceil \log_2 \sigma \rceil}$ is an integer then we have the following case. For example, if an alphabet is represented by two bits in the compressed file (i.e. $\sigma = 3$ or 4) then a pattern can only start at even positions in the text-blocks. So in this case we only need to consider expressions 1, 3, 5 and 7.

Suppose $\frac{8}{\lceil \log_2 \sigma \rceil}$ is not an integer then we have the following case. For example if we are using 3 bits (i.e. $5 \leq \sigma \leq 8$) to represent an alphabet, then we need all the eight expressions. But in any comparison we need at most three expressions. Consider three consecutive text-blocks. Without loss of generality assume that the binary representation of a character starting at position 0 in the first of these three blocks. Then a pattern can start at positions 0, 3, or 6 in the first text-block, positions 1,4 or 7 in the second text-block or positions 2 or 5 in the third text-block. For the first text-block we need to consider the expressions 7, 4 and 1. For the second text-block we need to consider the expressions 0, 3 and 6. For the third text-block we need to consider the expressions 2 and 5.

5 The average running time

The pre-processing of our algorithm takes $O(m)$ time, as the Block-Table, Order-Table and the Search-Table can be constructed in $O(m)$ time. The worst case for the search will take $O(mn)$ time. In this section we will show that the algorithm performs on average at most $2n$ comparisons. ¿From this we can say that the average running time of the algorithm is $O(n + m)$. We also justify this with experiments at the end of this section.

At the end of the previous section we showed that we need to consider all eight expressions only when $\sigma = 2$. First we prove that the average number of comparisons for this worst case.

There are only 256 possible different blocks. If we assume that each of the 256 blocks occurs in a text with equal frequency, then we have the following lemma. Let $\Gamma_{PB}(j)$ be the probability of a pattern-block j matches a text-block.

Lemma 1: $\Gamma_{PB}(j) = \frac{1}{2^{8-w}}$, where w is the number of wildcard character N in the pattern-block.

Recall that when we compare a text-block with a pattern-block, we choose a pattern-block (from each expression) which has the minimum number of possibilities of matching with a text-block (i.e. the pattern-block with minimum number of wildcard character N). If any of these pattern-blocks matches with the text-block, then we choose the pattern-block with the minimum number of wild cards among the remaining pattern-blocks in the expression. In an attempt, for each expression we repeat this step until either a full match or mismatch is found.

For example, consider the expressions for $m = 34$. Figure 2 shows the values of w in a pattern-block for each expression (pattern-block numbers are in brackets).

Exp0:	7 (0)	0 (8)	0 (16)	0 (24)	0 (32)	7 (40)
Exp1:	6 (1)	0 (9)	0 (17)	0 (25)	0 (33)	
Exp2:	5 (2)	0 (10)	0 (18)	0 (26)	1 (34)	
Exp3:	4 (3)	0 (11)	0 (19)	0 (27)	2 (35)	
Exp4:	3 (4)	0 (12)	0 (20)	0 (28)	3 (36)	
Exp5:	2 (5)	0 (13)	0 (21)	0 (29)	4 (37)	
Exp6:	1 (6)	0 (14)	0 (22)	0 (30)	5 (38)	
Exp7:	0 (7)	0 (15)	0 (23)	0 (31)	6 (39)	

Figure 2: The number of wildcards in pattern-blocks for $m = 34$

There are three columns with all zeros which are the first three columns in the Order-Table. In general, for all m, if $m \bmod 8 \neq 7$, there are $\lambda = \lfloor \frac{m-7}{8} \rfloor$ number of columns will have all zeros. If $m \bmod 8 = 7$, and $m \geq 15$ we will have $\lambda - 1$ columns with all zeros, and the remaining one with seven zeros in a column and the eighth zero in another column. For example, Figure 3 shows the number of wildcards in pattern-blocks for $m = 23$ (i.e. $m \bmod 8 = 7$.). We can see that there is one (i.e. $\lambda - 1$) column which is the second column with all zeros. The remaining column of all zeros is the fourth column with seven zeros and the eighth zero is in the first column (shown in bold font).

Exp0:	7 (0)	0 (8)	0 (16)	2 (24)
Exp1:	6 (1)	0 (9)	0 (17)	3 (25)
Exp2:	5 (2)	0 (10)	0 (18)	4 (26)
Exp3:	4 (3)	0 (11)	0 (19)	5 (27)
Exp4:	3 (4)	0 (12)	0 (20)	6 (28)
Exp5:	2 (5)	0 (13)	0 (21)	7 (29)
Exp6:	1 (6)	0 (14)	0 (22)	
Exp7:	**0 (7)**	0 (15)	1 (23)	

Figure 3: The number of wildcards in pattern-blocks for $m = 23$

From this observation we have Lemma 2. Let Φ_i be the probability of i number of pattern-blocks matching with the text-blocks in an expression at an attempt. In other words Φ_i is the probability of the algorithm making *at least* $i+1$ comparisons at an attempt.

Lemma 2: For all m and $\sigma = 2$, $1 \leq i \leq \lambda$, $\Phi_i = 8 \times \frac{1}{256^i}$, where $\lambda = \lfloor \frac{m-7}{8} \rfloor$.

Proof: For all m, each expression has λ number of pattern-blocks with $w = 0$. At an attempt, we can choose pattern-blocks with $w = 0$ from each of the eight expressions for the first λ comparisons. From Lemma 1 we have $\Gamma_{PB}(j) = 1/256$ if $w = 0$. In an attempt we will have the $i + 1^{th}$ comparison only if i number of pattern-blocks in an expression matches the corresponding text-blocks. The probability of i matches for an expression is $\frac{1}{256^i}$ and there are eight expressions and so Φ_i is $\frac{8}{256^i}$, $1 \leq i \leq \lambda$. □

In an attempt, for $2 \leq m \leq 9$ and $10 \leq m \leq 14$ we have at most 2 and 3 comparisons respectively. Hence we only need to know the values of Φ_1 for $2 \leq m \leq 9$, and Φ_1 and Φ_2 for $10 \leq m \leq 14$. We can calculate these values easily. For example, the following shows the values of w in a pattern-block for each expression (pattern-block numbers are in brackets) for $m = 10$. First we will select the pattern-blocks 8 to 11 and 4 to 7.

Exp0:	7	(0)	0	(8)	7	(16)
Exp1:	6	(1)	0	(9)		
Exp2:	5	(2)	1	(10)		
Exp3:	4	(3)	2	(11)		
Exp4:	3	(4)	3	(12)		
Exp5:	2	(5)	4	(13)		
Exp6:	1	(6)	5	(14)		
Exp7:	0	(7)	6	(15)		

Figure 4: The number of wildcards in pattern-blocks for $m = 10$

$$\begin{aligned}
\Phi_1 &= \Gamma_{PB}(8) + \Gamma_{PB}(9) + \Gamma_{PB}(10) + \Gamma_{PB}(11) + \Gamma_{PB}(4) + \Gamma_{PB}(5) + \Gamma_{PB}(6) \\
&\quad + \Gamma_{PB}(7) \\
&= \frac{1}{8^{8-0}} + \frac{1}{8^{8-0}} + \frac{1}{8^{8-1}} + \frac{1}{8^{8-2}} + \frac{1}{8^{8-3}} + \frac{1}{8^{8-2}} + \frac{1}{8^{8-1}} + \frac{1}{8^{8-0}} \text{ (Lemma 1)} \\
&= 1/256 + 1/256 + 1/128 + 1/64 + 1/32 + 1/64 + 1/128 + 1/256 \\
&= 23/256
\end{aligned}$$

For Φ_2 we only need to consider the first expression. We can have at least 3 comparisons, iff pattern-blocks 8 and (assume we select) 0 match with the corresponding text-blocks.

$$\begin{aligned}
\Phi_2 &= \Gamma_{PB}(8) \times \Gamma_{PB}(0) \\
&= \frac{1}{8^{8-0}} \times \frac{1}{8^{8-7}} \text{ (Lemma 1)} \\
&= 1/256 \times 1/2 \\
&= 1/512
\end{aligned}$$

In an attempt, for all $m \geq 15$, after λ comparisons the pattern-blocks which have not yet been compared will be similar to the expressions for patterns of length m',

$7 \leq m' \leq 14$, where $m' = (m \bmod 8) + 8$ if $m \bmod 8 \neq 7$. Otherwise $m' = 7$. In other words, if we remove all the λ columns with all zeros from the expressions of pattern length $m \geq 15$, the number of wildcards in pattern-blocks will be the same as in the expressions of pattern length m'. For example, if we remove (i.e. λ) columns of all zeros from the number of wildcards in pattern-blocks, for $m = 34$ (see Figure 2), we will get the number of wildcards in pattern-blocks, for $m' = 10$ (see Figure 4) as in Figure 5.

Exp0:	7 (0)	0 (32)	7 (40)
Exp1:	6 (1)	0 (33)	
Exp2:	5 (2)	1 (34)	
Exp3:	4 (3)	2 (35)	
Exp4:	3 (4)	3 (36)	
Exp5:	2 (5)	4 (37)	
Exp6:	1 (6)	5 (38)	
Exp7:	0 (7)	6 (39)	

Figure 5: The number of wildcards in pattern-blocks, for $m' = 10$

Note that in any attempt for all m, we can have at most $\lambda + 1$ matches before we make the last comparison, if $m \bmod 8 = 0$, 1 or 7, otherwise $\lambda + 2$. For $m > 15$, we need to know $\Phi_{\lambda+1}$ and $\Phi_{\lambda+2}$. From the above observation we can calculate these values from the values of Φ_1 and Φ_2 for m, $7 \leq m \leq 14$. From these base values we can have the following Lemma. Note that $\lambda = 0$ for all $m \leq 14$.

Lemma 3: For $m \geq 7$,

$$\begin{aligned} \Phi_{\lambda+1} &= (1/256)^{\lambda} \times \alpha_b \text{ and} \\ \Phi_{\lambda+2} &= (1/256)^{\lambda} \times \beta_b, \end{aligned}$$

where α_b and β_b are the values of b^{th} base case in the first and second columns in the table below respectively and $b = m \bmod 8$.

base case	α	β
0	11/64	
1	15/128	
2	23/256	1/512
3	1/16	1/512
4	13/256	1/512
5	5/128	3/2048
6	9/256	5/4096
7	7/32	

Let Ψ_i be the probability of making *exactly* i comparisons at an attempt. Using Φ_i we can have an equation for Ψ_i:

$$\Phi_i = \Psi_{i+1} + \Psi_{i+2} + \cdots$$

This gives

$$\Psi_i = \Phi_{i-1} - \Phi_i$$

We know that we will make at least one comparison in every attempt. So Φ_0 is 1.

For all m and $\sigma = 2$, the maximum number of comparisons in any attempt is $\mu = \lceil \frac{m+7}{8} \rceil$, which is equal to $\lambda + 2$ if $m \bmod 8 = 0$, 1 or 7, otherwise $\lambda + 3$. So Φ_i is 0 for all $i \geq \mu$. This gives:

$$\begin{aligned} \Psi_1 &= 1 - \Phi_1 \\ \Psi_i &= \Phi_{i-1} - \Phi_i, 2 \leq i \leq \mu - 1 \\ \Psi_\mu &= \Phi_{\mu-1} \end{aligned}$$

Lemma 4: For $\sigma = 2$, the total number of comparisons, Ψ_{Total}, is less than or equal to $2n'$ on average, where n' is the number of text-blocks in the text.

Proof: (1)

$$\begin{aligned} \Psi_{Total} &= n' \times \sum_{i=1}^{\mu} i \times \Psi_i \\ &= n' \times ((1 - \Phi_1) + 2(\Phi_1 - \Phi_2) + \cdots + \mu - 1(\Phi_{\mu-2} - \Phi_{\mu-1}) + \mu\Phi_{\mu-1}) \\ &= n' \times (1 + \Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}) \\ &= n' \times (1 + \sum_{i=1}^{\lambda} \frac{8}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2}) \text{ (Lemma 2)} \\ &\leq 2n' \end{aligned}$$

This is because $\sum_{i=1}^{\lambda} \frac{8}{256^i} + \Phi_{\lambda+1} + \Phi_{\lambda+2} \leq 1$ (Lemmas 2 and 3) □

Lemma 5: For $\sigma > 2$, the total number of comparisons, Ψ_{Total}, is O(n), where n is the size of the original text.

Proof: The probability of more than one comparison in an attempt is $\Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}$ (see Lemma 4), where $\mu = \lceil \frac{m\lceil \log_2 \sigma \rceil + 7}{8} \rceil$. Note that $m\lceil \log_2 \sigma \rceil$ is the length of the pattern when we convert it into a binary string. We show in the last section that in an attempt we only need to consider a maximum of $\lceil \frac{8}{\lceil \log_2 \sigma \rceil} \rceil$ expressions when $\sigma > 2$. Hence, for $\sigma > 2$, $\Phi_1 + \cdots + \Phi_{\mu-2} + \Phi_{\mu-1}$ is less than the value given for $\sigma = 2$. □

From these Lemmas we have the following Theorem.

Theorem: The average running time of our algorithm is O($n + m$).

To show this is also true in practice we counted the number of comparisons by running our algorithm. Table 2 in the Appendix shows the estimated number of comparisons (Ψ_{Total}) and the actual number of comparisons. We used the same texts for each σ as in Table 1 (Section 3). For each pattern length we use 100 random patterns. The actual number of comparisons in the table is the total number of comparisons divided by the number of patterns of that length. The pattern length given in Table 2 is the length of the original pattern.

6 Comparison with existing string matching algorithms

In this section we compare the existing string matching algorithms with our algorithm, the BRS algorithm. There are a number of strings matching algorithms available in the literature. We have chosen seven of them, BR, BM, HOR, QS, RAI, SMI, RF and NR algorithms which can be found in [1, 2, 7, 12, 10, 11, 6, 9] respectively. The first six algorithms were found to be fast in [1]. Animations of these algorithms can be found at [4] and more information about the algorithms can be found in [3].

The experiments were carried for all the algorithms on an un-compressed text, except for our BRS algorithm and the NR algorithm [9]. The text used for these experiments was the same text as in Table 1 (Section 3). The patterns used in these experiments are generated randomly. For each σ and m, we tested 100 patterns and we measured the total (user) time (including pre-computation time) in seconds to search for all 100 patterns. We repeat each test 10 times and take the average. We used an Intel 486-DX2-66 processor based machine with 8 megabytes of RAM and a 100 megabyte hard drive running S.u.S.E. Linux 5.2 to conduct the experiments. All the algorithms were coded in C. The results of the experiments are in the Appendix (Tables 4 to 8).

7 Conclusions

The method described in Section 2 to store a text will reduce the original text size to $\frac{\lceil \log_2 \sigma \rceil}{8} n$. Although this method is not compression as in the literature, it reduces the space and it is comparable with the existing methods.

The main aim of this paper is string matching in a compressed text. Our string matching algorithm compares two blocks, checks whether a prefix (or suffix) of a block is a suffix (or prefix) of the other block. This takes constant time and uses byte processing. In practice, byte processing is much faster than bit processing because bit shifting and masking operations are not necessary at search time. We prove that the average time taken by our algorithm is $O(n + m)$. We also justified our average running time by experiments.

Using our algorithm one can keep texts (with an alphabet of $2 \leq \sigma \leq 128$ characters) compressed indefinitely and perform the search for a pattern. These methods will save both time and space. The experimental results show that our algorithm is more efficient than the existing algorithms for $\sigma \leq 16$. Texts with such a small alphabet are DNA, RDNA and hexadecimal files. One can improve our algorithm so that it performs well for large alphabet sets.

References

[1] Berry T., Ravindran S., *"A fast string matching algorithm and experimental results"*, Prague Stringology Club Workshop '99, 1999.

[2] Boyer R. S., Moore J. S., *"A fast string searching algorithm"*, Communications of the ACM, 23(5), pp 1075-1091, 1977.

[3] Charras C., Lecroq T., 1997, Exact string matching, available at: `http://www-igm.univ-mlv.fr/~lecroq/string.ps`

[4] Charras C., Lecroq T., 1998, Exact string matching animation in JAVA available at: `http://www-igm.univ-mlv.fr/~lecroq/string/`

[5] Crochemore M., Rytter W., *"Text algorithms"*, Oxford University Press, 1994.

[6] Crochemore, M., Czumaj, A., Gasieniec, L., Jarominek, S., LeCroq, T., Plandowki, W., Rytter, W., *"Speeding up two string matching algorithms"*, Algorithmica 12(4/5), pp 247-267, 1994.

[7] Horspool R. N., *"Practical fast searching in strings"*, Software Practice and Experience, 10(6), pp 501-506, 1980.

[8] Huffman D.A., *"A method for the construction of minimum redundancy codes"*, Proceedings of the Institute of Radio Engineers, 40, pp 1098-1101, 1951.

[9] Navarro G., Raffinot M., *"A general practical approach to pattern matching over Ziv-Lempel compressed text"*, Proceedings of Combinatorial Pattern Matching 99, Lecture Notes in Computer Science, 1645, pp 14-36, 1999. .

[10] Raita T., *"Tuning the Boyer-Moore-Horspool string searching algorithm"*, Software Practice and Experience, 22(10), pp 879-884, 1992.

[11] Smith P. D., *"Experiments with a very fast substring search algorithm"*, Software Practice and Experience 21(10), pp 1065-1074,1991.

[12] Sunday D. M., *"A very fast substring search algorithm"*, Communications of the ACM. 33(8), pp 132-142, 1990.

[13] Welch T. A., *"A technique for high-performance data compression"*, IEEE Computer, 17(6), pp 8-19, 1984.

[14] Ziv J., Lempel A., *"A universal algorithm for sequential data compression"*, IEEE Trans. On Information Theory, IT-23, 1978.

Appendix

σ	Our method	Huffman	Compress	Gzip	NR
2	62500	62500	71579	79644	121110
3	125000	104107	110629	118776	178706
4	125000	125000	136945	146402	215764
5	187500	149935	161641	168813	244192
8	187500	187500	209053	211543	297634
9	250000	201313	223571	226617	310964
16	250000	250000	288546	285834	373658
17	312500	257293	294476	290854	377491
32	312500	312500	367527	330150	449265
33	375000	316232	370975	332592	451570
64	375000	375000	461069	378224	493981

Table 1: Compressed text sizes for a random text of 500,000 bytes.

alphabet of 2			alphabet of 4			alphabet of 8		
Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual
5	85938	85413	2	156250	156258	2	207031	255959
10	68237	68276	4	135742	136513	4	190795	191710
20	64556	64446	8	127288	126999	6	189632	189931
30	64460	64460	12	126962	126962	8	189462	189537
40	64460	64467	18	126960	126962	12	189460	189898
50	64460	64473	24	126960	126962	16	189460	189551

Table 2: Estimated versus actual number of comparisons of our BRS algorithm

alphabet of 16			alphabet of 32			alphabet of 64		
Pat. Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual	Pat Len.	Ψ_{Total}	Actual
2	260742	265331	2	318237	322155	2	378296	378678
4	252288	252013	3	314507	314567	3	377132	376990
6	251960	251956	4	314556	314581	4	376962	376980
8	251960	251962	6	314460	314509	5	376962	376965
10	251960	251957	8	314460	314297	7	376960	376482
12	251960	251959	10	314460	314348	9	376960	376503

Table 3: Estimated versus actual number of comparisons of our BRS algorithm

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
5	14.2	29.1	31.3	31.5	31.1	28.7	31.8	32.6	30.7
10	5.3	27.0	24.7	30.9	31.0	27.7	31.4	22.0	30.5
20	4.4	27.3	20.4	28.8	32.4	26.6	31.0	18.2	29.5
30	4.2	27.3	18.3	31.2	31.2	28.0	31.4	16.0	27.5
40	4.2	28.3	17.3	29.7	31.3	27.9	30.7	13.5	28.5
50	5.2	26.5	16.4	30.5	30.0	27.7	31.1	15.0	28.4

Table 4: Search times for $\sigma = 2$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
4	8.6	15.3	20.5	20.9	20.3	20.2	23.8	21.3	21.6
8	5.3	13.1	17.6	18.1	19.3	18.7	19.5	17.3	20.7
12	5.7	12.5	19.3	18.8	18.7	18.0	18.3	15.3	17.6
16	5.7	12.9	17.4	15.8	17.4	17.3	17.7	13.6	18.4
20	5.7	12.0	17.2	18.5	17.6	17.9	18.5	14.1	20.5
24	5.7	12.5	16.7	17.7	18.6	16.6	18.1	12.7	20.2

Table 5: Search times for $\sigma = 4$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
3	9.9	15.7	22.6	18.6	17.6	16.9	19.3	18.0	28.9
4	14.0	17.0	25.8	21.2	18.7	21.1	21.3	18.9	27.6
6	8.6	13.7	19.7	16.9	16.0	16.5	16.0	15.8	23.5
10	8.5	12.7	15.7	14.1	15.1	14.9	15.1	14.1	25.5
14	8.7	12.0	15.7	12.4	14.2	13.3	13.8	13.0	25.2
18	8.4	11.1	15.0	13.6	14.0	12.7	13.4	13.2	25.5

Table 6: Search times for $\sigma = 8$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
2	14.1	19.4	33.0	25.8	21.0	24.4	24.4	19.6	33.4
4	9.8	15.2	21.7	17.8	17.0	17.9	17.7	16.2	31.5
6	9.8	13.4	16.6	14.0	14.6	13.6	15.0	13.4	31.5
8	9.7	12.3	16.2	14.4	13.9	13.2	13.8	13.2	27.7
10	9.7	12.1	14.2	13.2	13.6	12.3	13.0	13.6	29.6
12	9.9	11.1	14.3	13.0	12.3	13.0	13.4	12.9	31.1

Table 7: Search times for $\sigma = 16$

Pat. Length	BRS	BR	BM	HOR	QS	RAI	SMI	RF	NR
2	66.5	18.6	33.0	23.6	19.5	23.8	22.3	19.0	39.1
3	42.1	16.3	24.0	20.2	17.7	19.8	20.3	16.6	40.1
4	31.9	14.8	21.2	17.1	15.5	15.4	17.5	15.0	37.2
6	39.7	12.3	17.5	13.2	14.7	14.6	15.5	14.1	36.2
8	37.9	12.3	15.5	13.4	13.4	13.2	13.9	13.5	38.8
10	48.2	11.5	15.0	12.4	11.8	12.5	13.7	13.0	34.2

Table 8: Search times for $\sigma = 32$

Approximate String Matching in Musical Sequences*

Maxime Crochemore[1], Costas S. Iliopoulos[2 †],
Thierry Lecroq[3] and Y. J. Pinzon[2 ‡]

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, France.
mac@univ-mlv.fr,
[2] Dept. of Computer Science, King's College London, London WC2R 2LS, England,
and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA.
Australia
{csi,pinzon}@dcs.kcl.ac.uk,
[3] LIFAR - ABISS, Université de Rouen, 76821 Mont Saint Aignan Cedex, France.
lecroq@dir.univ-rouen.fr

e-mail: T.Berry@livjm.ac.uk,S.Ravindran@livjm.ac.uk

Abstract. Here we consider computational problems on δ-approximate and (δ, γ)-approximate string matching. These are two new notions of approximate matching that arise naturally in applications of computer assisted music analysis. We present fast, efficient and practical algorithms for these two notions of approximate string matching.

Key words: String algorithms, approximate string matching, dynamic programming, computer-assisted music analysis.

1 Introduction

This paper focuses on a set of string pattern-matching problems that arise in musical analysis, and especially in musical information retrieval. A musical score can be viewed 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). Approximate repetitions in one or more musical works play a crucial role in discovering similarities between different musical entities and may be used for establishing "characteristic signatures" (see [6]). Such algorithms can be particularly useful for melody identification and musical retrieval.

The approximate repetition problem has been extensively studied over the last few years. Efficient algorithms for computing the approximate repetitions are directly applicable to molecular biology (see [7, 9, 12]) and in particular in DNA sequencing by

*This work was partially supported by a NATO grant PST.CLG.977017.
†Partially supported by a Marie Curie fellowship, Wellcome and Royal Society grants.
‡Partially supported by an ORS studentship and EPSRC Project GR/L92150.

hybridization ([13]), reconstruction of DNA sequences from known DNA fragments (see [15, 16]), in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species ([15]).

The approximate matching problem has been used for a variety of musical applications (see overviews in McGettrick [11]; Crawford et al [6]; Rolland et al [14]; Cambouropoulos et al [3]). It is known that exact matching cannot be used to find occurrences of a particular melody. Approximate matching should be used in order to allow the presence of errors. The number of errors allowed will be referred to as δ. This paper focuses in one special type of approximation that arise especially in musical information retrieval, i.e. δ-approximation. Most computer-aided musical applications adopt an absolute numeric pitch representation (most commonly MIDI pitch and pitch intervals in semitones; duration is also encoded in a numeric form). The absolute pitch encoding, however, may be insufficient for applications in tonal music as it disregards tonal qualities of pitches and pitch-intervals (e.g. a tonal transposition from a major to a minor key results in a different encoding of the musical passage and thus exact matching cannot detect the similarity between the two passages). One way to account for similarity between closely related but non-identical musical strings is to use what will be referred to as δ-approximate matching (and γ-approximate matching). In δ-approximate matching, equal-length patterns consisting of integers match if each corresponding integer differs by not more than δ- e.g. a C-major $\{60, 64, 65, 67\}$ and a C-minor $\{60, 63, 65, 67\}$ sequence can be matched if a tolerance $\delta = 1$ is allowed in the matching process (γ-approximate matching is described in the next section).

In [4], a number of efficient algorithms for δ-approximate matching were presented (i.e. the Shift-And algorithm and Shift-Plus algorithm). The Shift-And algorithm is based on the $O(1)$-time computation of different states for each symbol in the text. Hence the overall complexity is $O(n)$. These algorithms use the bitwise technique. It is possible to adapt fast and practical exact pattern matching algorithms to these kind of approximations. In this paper we will present the adaptations of the Tuned-Boyer-Moore [8], the Skip-Search algorithm [5] and the Maximal-Shift algorithm [17] and present some experiments to assert that these adaptations are faster than the algorithms using the bitwise technique.

The paper is organised as follows. In the next section we present some basic definitions for strings and background notions for approximate matching. In Sections 3-5 we present the adaptation of Tuned-Boyer-Moore, Skip-Search and Maximal-Shift algorithms to speed-up δ-approximate pattern matching algorithms and in section 6 to speed-up (δ, γ)-approximate pattern matching algorithms. In section 7 we present the experimental results of these algorithms. Finally in Section 8 we present our conclusions.

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 by Σ^*. A string x of length n is represented by $x_1 \ldots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$; we equivalently say that the string w occurs at position $|u| + 1$ of the string x. The position $|u| + 1$ is said to be the *starting position* of w in x and the position $|w|+|u|$ the *end position* of w in x. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$.

The string xy is a *concatenation* of two strings x and y. The concatenations of k copies of x is denoted by x^k. For two strings $x = x_1 \dots x_n$ and $y = y_1 \dots y_m$ such that $x_{n-i+1} \dots x_n = y_1 \dots y_i$ for some $i \geq 1$, the string $x_1 \dots x_n y_{i+1} \dots y_m$ is a *superposition* of x and y. We say that x and y *overlap*.

Let x be a string of length n. The integer p is said to be a *period* of x, if $x_i = x_{i+p}$ for all $1 \leq i \leq n-p$. *The period* of a string x is the smallest period of x. A string y is a *border* of x if y is a prefix and a suffix of x.

Let Σ be an alphabet of integers and δ an integer. Two symbols a, b of Σ are said to be δ-approximate, denoted $a \stackrel{\delta}{=} b$ if and only if

$$|a-b| \leq \delta$$

We say that two strings x, y are δ-approximate, denoted $x \stackrel{\delta}{=} y$ if and only if

$$|x| = |y|, \text{ and } x_i \stackrel{\delta}{=} y_i, \ \forall i \in \{1, \dots, |x|\} \tag{2.1}$$

For a given integer γ we say that two strings x, y are γ-approximate, denoted $x \stackrel{\gamma}{=} y$ if and only if

$$|x| = |y|, \text{ and } \sum_{i=1}^{|x|} |x_i - y_i| \leq \gamma \tag{2.2}$$

Furthermore, we say that two strings x, y are $\{\gamma, \delta\}$-approximate, denoted $x \stackrel{\delta,\gamma}{=} y$, if and only if x and y satisfy conditions (2.1) and (2.2).

3 δ-Tuned-Boyer-Moore Approximate Pattern Matching

The problem of *δ-approximate pattern matching* is formally defined as follows: given a string $t = t_1 \dots t_n$ and a pattern $p = p_1 \dots p_m$ compute all positions j of t such that

$$p \stackrel{\delta}{=} t[j..j+m-1]$$

A naive solution of this problem is to build an Aho-Corasick automaton (see [1]) of all strings that are δ-approximate to p and then use the automaton to process t. The time required to build the automaton is $O(|\Sigma|^\delta)$, thus this method is of no practical use as e.g we can have $|\Sigma| \approx 180$ and $|\delta| \approx 10$. In [4] an efficient algorithm was presented based on the $O(1)$-time computation of the "delta states" by using bit operations under the assumption that $m \leq w$, where w is the number of bits in a machine word.

Here we present an adaptation of the Tuned-Boyer-Moore for exact pattern matching algorithm to δ-approximate pattern matching. The exact pattern matching problem consists in finding one or more (generally all) exact occurrences of a pattern p of length m in a text t of length n. Basically a pattern matching algorithm uses a window which size is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then it checks if the pattern occurs in the window and shifts the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text.

The TUNED-BOYER-MOORE algorithm is a very fast practical variant of the famous BOYER-MOORE algorithm [2]. It only uses the occurrence shift function to perform the shifts. The occurrence shift function is defined for each symbol a in the alphabet Σ as follows:

$$shift[a] = \min\{\{m - i \mid p_i = a\} \cup \{m\}\}$$

The TUNED-BOYER-MOORE algorithm gains its efficiency by unrolling three shifts in a very fast skip loop to locate the occurrences of the rightmost symbol of the pattern in the text. Once an occurrence of p_m is found, it checks naively if the whole pattern occurs in the text. Then the shift consists in aligning the rightmost symbol of the window with the rightmost reoccurrrence of p_m in $p_1 \ldots p_{m-1}$, if any. The length s of this shift is defined as follows:

$$s = \min\{\{m - i \mid p_i = p_m \text{ and } i > 0\} \cup \{m\}\}$$

To do δ-approximate pattern matching, the shift function can be defined to be for each symbol a in the alphabet Σ the distance from the right end of the pattern of the closest symbol p_i such that $p_i \stackrel{\delta}{=} a$:

$$shift[a] = \min\{\{m - i \mid p_i \stackrel{\delta}{=} a\} \cup \{m\}\}$$

Then the length of the shift s becomes:

$$s = \min\{\{m - i \mid p_i \stackrel{2\delta}{=} p_m \text{ and } i > 0\} \cup \{m\}\}$$

The pseudo-code for δ-TUNED-BOYER-MOORE algorithm can be found in Figure 1.

4 δ-SKIP-SEARCH Approximate Pattern Matching

In the SKIP-SEARCH algorithm, for each symbol of the alphabet, a bucket collects all of that symbol's positions in p. When a symbol occurs k times in the pattern, there are k corresponding positions in the symbol's bucket. When the word is much shorter than the alphabet, many buckets are empty. The buckets are stored in a table z defined as follows:

$$z[a] = \{i \mid p_i = a\}$$

The main loop of the search phase consists of examining every mth text symbol, t_j (so there will be n/m main iterations). Then for t_j, it uses each position in the bucket $z[t_j]$ to obtain a possible starting point of p in t and checks if the pattern occurs at that position.

To do δ-approximate pattern matching, the buckets can be computed as follows:

$$z[a] = \{i \mid p_i \stackrel{\delta}{=} a\}$$

Figure 2 shows the pseudo-code for δ-SKIP-SEARCH algorithm.

```
δ-Tuned-Boyer-Moore(p, m, t, n, δ)
▷ Preprocessing
for all a ∈ Σ
      do shift[a] ← min{{m − i | p_i =^δ a} ∪ {m}}
s ← min{{m − i | p_i =^{2δ} p_m} ∪ {m}}
t_n ... t_{n+m−1} ← (p_m)^m
▷ Searching
j ← m
while j ≤ n
      do k ← shift[t_j]
          while k ≠ 0
               do j ← j + k
                   k ← shift[t_j]
                   j ← j + k
                   k ← shift[t_j]
                   j ← j + k
                   k ← shift[t_j]
          if p_1 ... p_{m−1} =^δ t_{j−m+1} ... t_{j−1} and j ≤ n
             then Report(j − m + 1)
          j ← j + s
```

Figure 1: Adaptation of the Tuned-Boyer-Moore exact pattern matching algorithm to do δ-approximate pattern matching.

5 δ-Maximal-Shift Approximate Pattern Matching

Sunday [17] designed an exact string matching algorithm where the pattern positions are scanned from the one which will lead to a larger shift to the one which will lead to a shorter shift, in case of a mismatch. Doing so one may hope to maximize the lengths of the shifts and thus to minimize the overall number of comparisons.

Formally we define a permutation

$$\sigma : \{1, 2, \ldots, m, m+1\} \to \{1, 2, \ldots, m, m+1\}$$

and a function *shift* such that

$$\mathit{shift}[\sigma(i)] \stackrel{\gamma}{=} \mathit{shift}[\sigma(i+1)]$$

for $1 \le i < m$ and

$$\mathit{shift}[\sigma(i)] = \min\{\ell \mid \text{ for } 1 \le j < i, p_{\sigma(j)-\ell} = p_{\sigma(j)} \text{ and } p_{\sigma(i)-\ell} \neq p_{\sigma(i)}\}$$

for $1 \le i \le m$ and $\sigma(m+1) = m+1$. Furthermore $\mathit{shift}[m+1]$ is set with the value of the period of the pattern p.

We also define a function bc for each symbol of the alphabet:

δ-SKIP-SEARCH(p, m, t, n, δ)
$\triangleright$ *Preprocessing*
for all $a \in \Sigma$
  do $z[a] \leftarrow \{i \mid p_i \overset{\delta}{=} a\}$
$\triangleright$ *Searching*
$j \leftarrow m$
while $j \leq n$
  do for all $i \in z[t_j]$
    do if $p \overset{\delta}{=} t_{j-i} \dots t_{j-i+m-1}$
      then REPORT$(j - i)$
  $j \leftarrow j + m$

Figure 2: Adaptation of the SKIP-SEARCH exact pattern matching algorithm to do δ-approximate pattern matching.

$$bc[a] = \begin{cases} \min\{j \mid 0 \leq j < m \text{ and } p_{m-j} = a\} & \text{, if } a \text{ occurs in } p \\ m & \text{, otherwise} \end{cases}$$

for $a \in \Sigma$.

Then, when the pattern is aligned with the $t[j..j+m-1]$ the comparisons are performed in the following order $\sigma(1), \sigma(2), \dots, \sigma(m)$ until the whole pattern is scanned or a mismatch is found. If a mismatch is found when comparing $p[\sigma(i)]$ then a shift of length $\max\{\mathit{shift}[\sigma(i)], bc[t[j+m+1]]\}$ is performed. Otherwise an occurrence of the pattern is found and the length of the shift is equal to the maximum value between the period of the pattern and $bc[t[j+m+1]]$. Then the comparisons resume with $p_{\sigma(1)}$ without keeping any memory of the comparisons previously done.

To perform δ-approximate string matching the two functions can be redefined as follows:

$$\mathit{shift}[\sigma(i)] = \min\{\ell \mid \text{ for } 1 \leq j < i, p_{\sigma(j)-\ell} =_{2\delta} p_{\sigma(j)} \text{ and } p_{\sigma(i)-\ell} \neq_{\delta} p_{\sigma(i)}\}$$

for $1 \leq i \leq m$ and

$$\mathit{shift}[m+1] = \min\{\ell \mid p[i] =_{2\delta} p[i+\ell] \text{ for } 1 \leq i \leq m-\ell\}$$

and

$$bc[a] = \begin{cases} \min\{j \mid 0 \leq j < m \text{ and } p_{m-j} =_{\delta} a\} & \text{, if such a } j \text{ exists} \\ m & \text{, otherwise} \end{cases}$$

for $a \in \Sigma$.

The preprocessing phase can be done in $O(m^2)$. Figure 3 gives the pseudo-code of the searching phase.

```
δ-MAXIMAL-SHIFT(p, m, t, n, δ)
▷ Searching
j ← 0
while j ≤ n − m
     do i ← 1
        while i ≤ m and p[σ(i)] = t[j + σ(i)]
              do i ← i + 1
        if i > m
          then REPORT(j)
        j ← j + max{shift[σ(i)], bc[t[j + m + 1]]}
```

Figure 3: Adaptation of the MAXIMAL-SHIFT exact pattern matching algorithm to do δ-approximate pattern matching.

6 (δ, γ)-Approximate String Matching Algorithms

The problem of (δ, γ)-*approximate pattern matching* is formally defined as follows: given a string $t = t_1 \ldots t_n$ and a pattern $p = p_1 \ldots p_m$ compute all positions j of t such that

$$p \stackrel{\delta,\gamma}{=} t[j..j + m - 1]$$

In [4] this problem was solved by making use of the SHIFT-AND algorithm to find the δ-approximate matches of the pattern p in t. Once a δ-approximate match was found, it was then tested to check whether it is also a γ-approximate match. This was done by computing successive “delta states” and “gamma states” in $O(1)$ time using bit operations under the assumption that $m \leq w$ where w is the number of bits in a machine word.

In order to adapt the δ-TUNED-BOYER-MOORE, δ-SKIP-SEARCH and δ-MAXIMAL-SHIFT algorithms to the case of (δ, γ)-approximation, it just suffices to adapt the naive check of the pattern. The resulting algorithms are named (δ, γ)-TUNED-BOYER-MOORE algorithm, (δ, γ)-SKIP-SEARCH algorithm and (δ, γ)-MAXIMAL-SHIFT algorithm.

7 Experimental results

We implemented in C, in a homogeneous way, the following algorithms: SHIFT-AND, δ-TUNED-BOYER-MOORE, δ-SKIP-SEARCH, δ-MAXIMAL-SHIFT, SHIFT-PLUS, (δ,γ)-TUNED-BOYER-MOORE, (δ, γ)-SKIP-SEARCH and (δ, γ)-MAXIMAL-SHIFT.

We randomly built a text of 500k symbols on an alphabet of size $|\Sigma| = 70$. We then searched for each values 100 patterns and took the average running time. Times are measured in hundredth of seconds and include both preprocessing and searching times.

The results for δ-approximation are shown in tables 1 to 5. For the values used in these experiments, the δ-TUNED-BOYER-MOORE algorithm is always faster than the δ-SKIP-SEARCH algorithm which is itself always faster than the SHIFT-AND algorithm.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	32.98	10.78	18.61
9	32.90	10.55	18.11
10	32.93	10.10	17.65
20	32.86	9.32	15.81

Table 1: Running times for δ-approximation with $\delta = 5$.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	33.07	13.40	21.66
9	32.90	13.00	20.94
10	32.93	12.64	20.49
20	32.92	11.97	18.81

Table 2: Running times for δ-approximation with $\delta = 6$.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	33.65	16.65	24.99
9	33.14	16.05	24.06
10	33.05	15.71	23.62
20	32.93	14.82	21.42

Table 3: Running times for δ-approximation with $\delta = 7$.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	34.72	21.18	29.15
9	33.41	20.03	27.64
10	33.07	19.12	26.85
20	32.81	18.20	24.41

Table 4: Running times for δ-approximation with $\delta = 8$.

The results for (δ, γ)-approximation are shown in tables 6 to 10. For the values used in these experiments, the (δ, γ)-Tuned-Boyer-Moore algorithm is always faster than the (δ, γ)-Skip-Search algorithm which is itself always faster than the Shift-Plus algorithm.

Experiments conduct only on γ-approximation show that an adaptation to this case of the Skip-Search algorithm is faster than an adaptation of the Tuned-Boyer-Moore algorithm.

m	Shift-And	δ-Tuned-Boyer-Moore	δ-Skip-Search
8	36.46	26.82	34.64
9	34.46	24.36	31.46
10	33.41	23.61	30.55
20	33.00	22.32	27.54

Table 5: Running times for δ-approximation with $\delta = 9$.

m	Shift-Plus	(δ, γ)-Tuned-Boyer-Moore	(δ, γ)-Skip-Search
8	50.73	23.33	31.93
9	50.32	27.78	35.52
10	51.79	33.76	39.45
20	50.26	32.46	36.91

Table 6: Running times for (δ, γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 14$.

m	Shift-Plus	(δ, γ)-Tuned-Boyer-Moore	(δ, γ)-Skip-Search
8	50.88	23.16	31.99
9	50.86	28.70	36.40
10	51.87	33.74	39.58
20	51.11	32.53	37.38

Table 7: Running times for (δ, γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 15$.

m	Shift-Plus	(δ, γ)-Tuned-Boyer-Moore	(δ, γ)-Skip-Search
8	50.72	23.33	32.02
9	50.70	27.96	35.65
10	51.94	33.88	40.00
20	51.35	33.20	37.03

Table 8: Running times for (δ, γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 16$.

One should notice that the Shift-And and Shift-Plus algorithms need constant time to run whatever the values of the parameters are. In case of very high values for δ and/or γ they have to be considered as the best choice.

m	Shift-Plus	(δ,γ)-Tuned-Boyer-Moore	(δ,γ)-Skip-Search
8	50.67	23.29	32.20
9	50.83	28.38	35.74
10	51.93	34.41	39.91
20	50.18	32.94	37.10

Table 9: Running times for (δ,γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 17$.

m	Shift-Plus	(δ,γ)-Tuned-Boyer-Moore	(δ,γ)-Skip-Search
8	51.24	23.57	32.22
9	50.31	28.33	35.73
10	51.83	34.36	40.15
20	49.97	32.77	37.03

Table 10: Running times for (δ,γ)-approximation with $\delta = \min\{m, 10\}$ and $\gamma = 18$.

8 Conclusions

Here we presented the Skip-Search, Tuned-Boyer-Moore and Maximal-Shift approximate string matching algorithms that outperform the one presented in [4].

References

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

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

[3] E. Cambouropoulos, T. Crawford and C.S. Iliopoulos, (1999) Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects. In Proceedings of the AISB'99 Convention (Artificial Intelligence and Simulation of Behaviour), Edinburgh, U.K., pp. 42-47 (1999).

[4] E. Cambouropoulos, M. Crochemore, C. S. Iliopoulos, L. Mouchard, and Y. J. Pinzon. Algorithms for computing approximate repetitions in musical sequences. In R. Raman and J. Simpson, editors, *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 129–144, Perth, WA, Australia, 1999.

[5] C. Charras, T. Lecroq, and J.D. Pehoushek. A very fast string matching algorithm for small alphabets and long patterns. In M. Farach-Colton, editor, *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*,

number 1448 in Lecture Notes in Computer Science, pages 55–64, Piscataway, NJ, 1998. Springer-Verlag, Berlin.

[6] T. Crawford, C. S. Iliopoulos, R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition, *Computing in Musicology*, Vol 11 (1998) 73–100.

[7] V. Fischetti, G. Landau, J.Schmidt and P. Sellers, Identifying periodic occurences of a template with applications to protein structure, *Proc. 3rd Combinatorial Pattern Matching*, Lecture Notes in Computer Science, vol. 644, 1992, pp. 111–120.

[8] A. Hume and D. M. Sunday. Fast string searching. *Software–Practice and Experience*, 21(11):1221–1248, 1991.

[9] S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung, Efficients algorithms for molecular sequences analysis, *Proc. Natl. Acad. Sci., USA (1988) 85:841–845*

[10] G. Main and R. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *Journal of Algorithms* **5** (1984), pp. 422–432.

[11] P. McGettrick, MIDIMatch: Musical Pattern Matching in Real Time. MSc Dissertation, York University, U.K. (1997).

[12] A. Milosavljevic and J. Jurka, Discovering simple DNA sequences by the algorithmic significance method, *Comput. Appl. Biosci. (1993) 9:407–411*

[13] P. A. Pevzner and W. Feldman, Gray Code Masks for DNA Sequencing by Hybridization, *Genomics*, 23, 233-235 (1993).

[14] P.Y. Rolland, J.G. Ganascia, Musical Pattern Extraction and Similarity Assessment. In Readings in Music and Artificial Intelligence. E. Miranda. (ed.). Harwood Academic Publishers (forthcoming) (1999).

[15] J. P. Schmidt, All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings, *SIAM Journal on Computing* 27, 4 (1998), 972-992.

[16] S. S. Skiena and G. Sundaram, Reconstructing strings from substrings, *J. Computational Biol.* 2 (1995) 333–353.

[17] D. M. Sunday, A very fast substring search algorithm, *CACM*, Vol 33, (1990), pp. 132–142.

[18] S. Wu and U. Manber, Fast text searching allowing errors, *CACM*, Vol 35, (1992), pp. 83–91.

Construction of the CDAWG for a Trie

Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara,
Masayuki Takeda, and Setsuo Arikawa

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

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

Abstract. Trie is a tree structure to represent a set of strings. When the strings have many common prefixes, the number of nodes in the trie is much less than the total length of the strings. In this paper, we propose an algorithm for constructing the Compact Directed Acyclic Word Graph for a trie, which runs in linear time and space with respect to the number of nodes in the trie.

Key words: Pattern matching, Index structure, Trie, Suffix trie, Suffix tree, DAWG, CDAWG, Linear time algorithm

1 Introduction

Crochemore and Vérin displayed a relationship among suffix tries, suffix trees, Directed Acyclic Word Graphs (DAWGs), and Compact Directed Acyclic Word Graphs (CDAWGs) [CV97]. It states that a suffix tree (DAWG, resp.) can be obtained by compacting (minimizing, resp.) the corresponding suffix trie. Similarly, a CDAWG can be obtained by either compacting the corresponding DAWG or minimizing the corresponding suffix tree.

It is known that all of these indexing structures, except suffix tries, can be constructed in linear time and space: Weiner [Wei73], McCreight [McC76], and Ukkonen [Ukk95] for suffix trees, and Blumer et al. [BBH+85] for DAWGs.

In [BBH+87] Blumer et al. gave an algorithm for constructing a CDAWG by compacting the corresponding DAWG. Direct construction of a CDAWG from a given string is also important, since the hidden constant of the space complexity of CDAWGs is strictly smaller than those of suffix trees and DAWGs [BBH+87]. Actually, Crochemore and Vérin [CV97] gave the first algorithm that directly constructs CDAWGs, which is based on McCreight's algorithm for suffix trees. Recently, Inenaga et al. [IHS+01a] developed an on-line algorithm for the direct construction of CDAWGs, which is based on Ukkonen's algorithm.

Their algorithm can also construct a CDAWG for a set S of strings in linear time with respect to the total length ℓ of the strings in S. In this paper, we consider the case that the set S is given in the form of a trie, as input. Since the trie shares common prefixes of the strings in S, the number n of nodes of the trie is less than ℓ. We show a non-trivial extension of the algorithm that constructs CDAWG for a trie in $O(n)$ time and space.

Some related work can be seen in literature: Kosaraju [Kos89] introduced the suffix tree for a *reversed* trie, and showed an algorithm to construct it in $O(n \log n)$

time. Breslauer [Bre98] reduced it to $O(n)$ time. On the other hand, our algorithm constructs a CDAWG for a (normal) trie. We remark that our algorithm can be easily adopted to construct suffix trees and DAWGs instead of CDAWGs, with a slight modification in the same way as in [IHS⁺01b]. That means, all of these indexing structures for a trie can be constructed in linear time with respect to the number of nodes in the trie.

2 Preliminaries

The Compact Directed Acyclic Word Graph (CDAWG) can be seen as either the compaction of the Directed Acyclic Word Graph (DAWG), or the minimization of the suffix tree [BBH⁺87, CV97]. In this section, we recall the properties of CDAWGs, compared with those of suffix trees.

2.1 Notations

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 the 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, $|\varepsilon| = 0$. The ith symbol of a string w is denoted by $w[i]$ for $1 \le i \le |w|$, and the factor of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \le i \le j \le |w|$. For convenience, let $w[i:j] = \varepsilon$ for $j < i$.

Given a set S of strings, let $\|S\|$ represent the total length of the strings in S, and $|S|$ the cardinality of S. The sets of prefixes, factors, and suffixes of the strings in S are denoted by $\mathit{Prefix}(S)$, $\mathit{Factor}(S)$, and $\mathit{Suffix}(S)$, respectively.

2.2 Compact Directed Acyclic Word Graphs

Here we show the properties of CDAWGs. For comparison, we first recall the properties of suffix trees. The suffix tree for a set S of strings is a rooted tree whose edges are labeled with strings in $\mathit{Factor}(S)$ (see Fig. 1). We denote by $\mathit{STree}(S)$ the suffix tree for S. We here assume that each string w_i in $S = \{w_1, \ldots, w_k\}$ ends with a unique *endmarker* $\$_i \notin \Sigma$, where $1 \le i \le k$. Let $S' = \{w_1\$_1, \ldots, w_k\$_k\}$. On the above assumption, every string in $\mathit{Suffix}(S')$ is associated with a leaf node in $\mathit{STree}(S')$. $\mathit{STree}(S')$ has the following properties:

1. It has a *root* node, at most $\|S\| - 1$ *internal* nodes, and $\|S\| + |S|$ *leaf* nodes.

2. The root node and any internal nodes have at least two outgoing edges.

3. Labels of any two edges leaving the same node do not begin with the same letter.

4. Any string in $\mathit{Factor}(S)$ is represented by a path starting at the root node.

5. Any string in $\mathit{Suffix}(S')$ is represented by a path starting at the root node and ending at a leaf node.

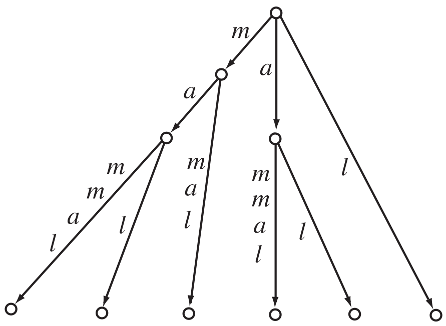

Figure 1: *STree*(S) for $S = \{mammal\}$. Since the last letter l plays the role of the endmarker, the endmarker is omitted.

The compact directed acyclic word graph (CDAWG) was first introduced by Blumer et al. [BBH+87]. The CDAWG for a set S of strings is a directed acyclic graph with edges labeled by strings in *Factor*(S) (see Fig. 2). We denote by *CDAWG*(S) the CDAWG for S. We similarly assume that each string in S ends with a unique endmarker. It is because the algorithm for constructing the CDAWG for a set of strings, which was given in [IHS+01a], requires the endmarkers. Remark that for any set S of strings *CDAWG*(S'), whose edges labeled with $\$_i$ are removed for any i $(1 \leq i \leq |S|)$, is equal to the CDAWG of S. *CDAWG*(S') has the following properties:

1. It has an *initial* node, at most $\|S\|-1$ *internal* nodes, and $|S|$ *final* nodes.
2. The initial node and any internal nodes have at least two outgoing edges.
3. Labels of any two edges leaving the same node do not begin with the same letter.
4. Any string in *Factor*(S) is represented by a path starting at the initial node.
5. Any string in *Suffix*(S') is represented by a path starting at the initial node and ending at a final node.
6. Suppose that a path spelling out $\alpha \in \Sigma^*$ ends at a node v. If a string β is always preceded by $\gamma \in \Sigma^*$ and $\alpha = \gamma\beta$ in any string $x \in S'$ such that $\beta \in Factor(x)$, the path spelling out β also ends at node v.

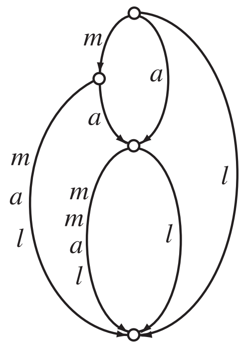

Figure 2: *CDAWG*(S) for $S = \{mammal\}$. Since the last letter l plays the role of the endmarker, the endmarker is omitted.

In Fig. 2, one can see that the path spelling out a ends at the same node as the one spelling out ma, with regard to the property 6 written above. This is because a is always preceded by m in string *mammal*.

Hereafter, we denote by (u, α, v) an edge labeled with α which starts at node u and ends at node v, both in CDAWGs and suffix trees.

2.3 Trie and Reversed Trie

Given a set $S = \{w_1, \ldots, w_k\}$ such that $w_i \notin \mathit{Suffix}(w_j)$ for any $1 \leq i \neq j \leq k$, consider the set $S'' = \{w_1\$ \ldots w_k\$\}$ where \$ denotes the common endmarker. Then the *reversed trie* for S'' is a tree in which strings in $\mathit{Suffix}(S'')$ are merged as long and many as possible [Bre98] (see Fig. 3). We associate each node in a reversed trie with a unique number, as in Fig. 3. We write as $\mathit{Trie}^R(S'')$ the reversed trie for a set S'' of strings. Every string in $\mathit{Prefix}(S'')$ is represented by a path beginning from a leaf node. The number of nodes in $\mathit{Trie}^R(S'')$ is at most $\|S''\| - |S''| + 2$. If the strings in S'' have long and many common suffixes, the number of nodes in $\mathit{Trie}^R(S'')$ is by far smaller than the upper bound $\|S''\| - |S''| + 2$.

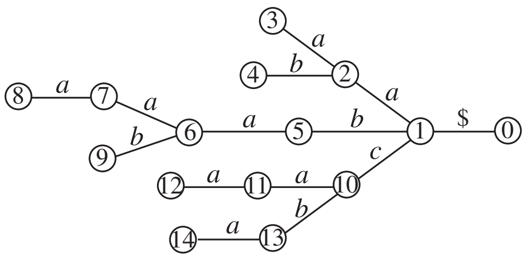

Figure 3: $\mathit{Trie}^R(S'')$ for $S'' = \{aaab\$, aac\$, aa\$, abc\$, bab\$, ba\$\}$

On the other hand, for a set $S' = \{w_1\$_1, \ldots, w_k\$_k\}$ where $\$_i$ denotes the endmarker for w_i $(1 \leq i \leq k)$, the *trie* for a set S' of strings is a tree in which strings in $\mathit{Prefix}(S')$ are merged as long and many as possible (see Fig. 4). We denote the trie for a set S' by $\mathit{Trie}(S')$. It is easy to see that the number of nodes in $\mathit{Trie}(S')$ is at most $\|S'\| + 1$. Thanks to the unique endmarkers, tries do not require the condition that reversed tries instead do. That is, even if a string $x \in S'$ belongs to $\mathit{Prefix}(y)$ for some string $y \in S'$, the path spelling out x always ends at a leaf node in $\mathit{Trie}(S')$. For example, although string aa is a prefix of $aaab$ in Fig. 4, the path spelling out $aa\$_3$ ends at leaf node 8.

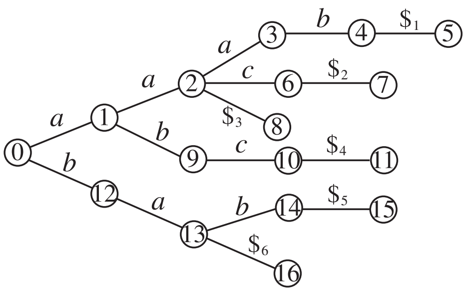

Figure 4: $\mathit{Trie}(S')$ for $S' = \{aaab\$_1, aac\$_2, aa\$_3, abc\$_4, bab\$_5, ba\$_6\}$

Tries are used as inputs of our algorithm that will be introduced in Section 4.

3 Algorithm to Construct the CDAWG for a Set of Strings

This section is devoted to recalling the algorithm to construct the CDAWG for a set of strings, which was proposed in [IHS+01a]. By illustrating the construction of *CDAWG*(*ababcbc*\$) in Fig. 5, we roughly show how the algorithm builds a CDWAG. More detailed description of the algorithm can be seen in [IHS+01b]. For simplicity, we have put a single string to the input of the algorithm in Fig. 5.

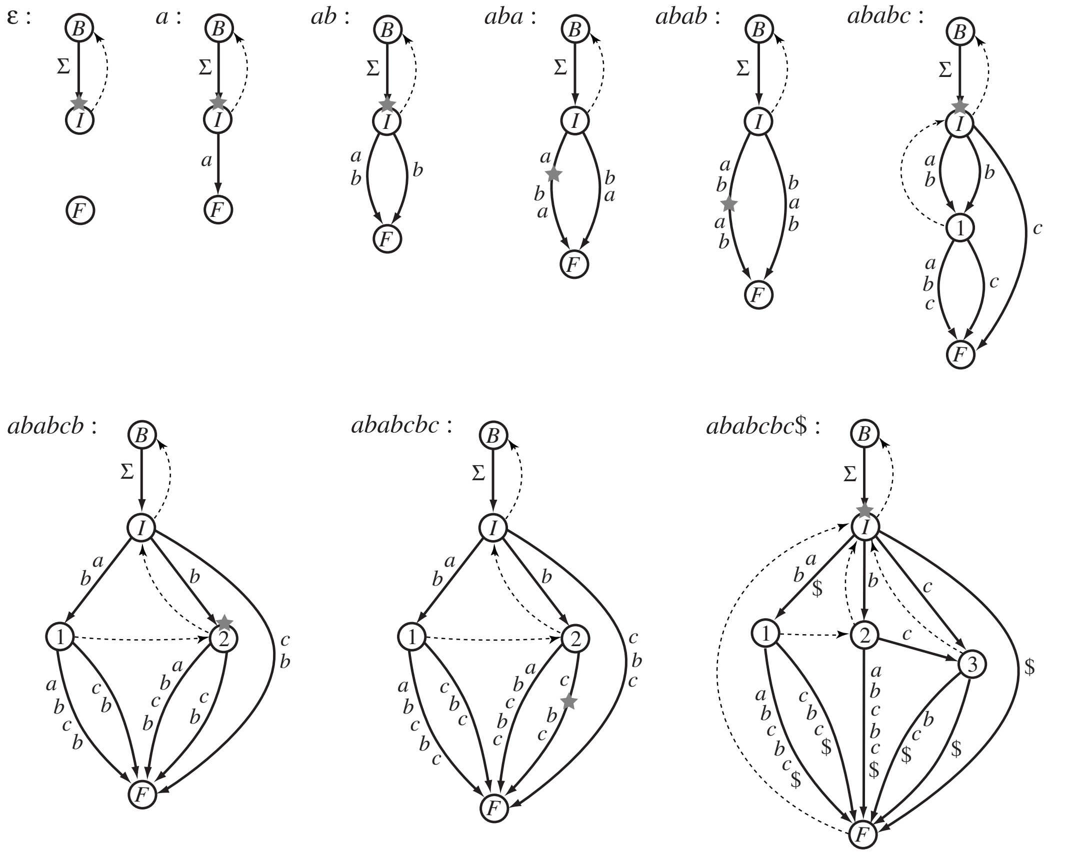

Figure 5: Construction of *CDAWG*(w) for w = *ababcbc*\$. The broken lines represent the suffix links, and the gray starred points represent the active points.

3.1 Suffix Links

As in literature [Wei73, McC76, Ukk95] about the construction of suffix trees, this algorithm to construct CDAWGs also utilizes *suffix links*. By means of the suffix links, the time complexity of these algorithms can be linear.

Let us assume that in a CDAWG the *shortest* path from the initial node to a node v spells out $\alpha = c\beta$, where $\alpha, \beta \in \Sigma^*$ and $c \in \Sigma$. Then, node v has the suffix link that points to a node u such that the path spelling out β is the longest path from the initial node to the node u. The algorithm determines the suffix link of each node during the construction of a CDAWG. For example, at phase *ababc* in Fig. 5, one can see that the suffix link of node 1, at which the path spelling out b ends, accordingly points to the initial node the empty string ε corresponds to. The suffix link of the initial node points to a special node, the *bottom* node, as in [Ukk95]. It has an edge labeled with Σ that represents an arbitrary letter in the alphabet Σ in this case. With this bottom node, we do not need to treat the initial node as an exception during the construction of a CDAWG. In Fig. 5, the bottom node, the initial node, and the final node are expressed by B, I, and F, respectively. Until the construction of the CDAWG of the whole input string is finished, the suffix link of the final node is left undefined (see the 1st phase to 8th in Fig 5), since the node to which the suffix link of the final node points can change during the construction. This implies that we cannot achieve a linear time algorithm if we update the suffix link of the final node at every phase. It can happen that a node r, which was the latest created in some phase, does not have the suffix link until another node is newly created in the phase, because the new node is just the one to which the suffix link of r should point.

3.2 The Active Point

The gray starred point in Fig. 5 denotes the location from which the algorithm starts to update the current CDAWG at the next phase. This is called *active point* similarly in [Ukk95]. An active point p is represented by the pair of a node v and a string α such that p can be reached from node v along the edge whose label begins with α. In the running example, the active point is represented by $(2, \varepsilon)$ at phase *ababcb*, whereas it is represented by $(2, c)$ at phase *ababcbc*.

From now on, we sketch how the active point moves and stops in a phase. Suppose that the algorithm has already finished a phase γ and now faces phase γc with $c \in \Sigma$, that is, a letter c follows γ in the input string. Then there can be four cases, that is, the active point is now:

(1) on a node that has an outgoing edge whose label begins with letter c;

(2) on a node that has no outgoing edge whose label begins with letter c;

(3) on the middle of an edge and followed by letter c in the label of the edge;

(4) on the middle of an edge and not followed by letter c in the label of the edge.

In case (1), the active point advances by letter c along the edge and then stops over there. This can be seen between phase *ab* and phase *aba* in Fig. 5. If it faces case (3) in the following phases, the active point keeps on moving and stopping along the edge, as seen in phase *abab*. In case (2), a new edge labeled with c is created and then connected from the node the active point is now on to the final node. The active point then moves to another node via the suffix link in order to check if there is an outgoing edge whose label begins with letter c. Finally, in case (4), a new node is created where the active point presents, splits the edge into two over there, and creates a new edge labeled with c from the new node to the final node. Then the algorithm has to look for the location where the active point will move next.

We illustrate how the algorithm behaves after facing case (4). See phase *abab* and phase *ababc* in Fig. 5. Since the active point of phase *abab* can not move along the edge any longer because c dose not follow *ab* there, new node 1 is created and then a new edge labeled with c is created and connected to the final node. After that, the algorithm has to find the location where the active point next moves. Since node 1 does not have the suffix link yet, the active point moves backwards to node I that has the suffix link. After arriving at node B via the suffix link of node I, it resumes moving along the path corresponding to ab, where ab is the part of the label of the edge that the active point moved backwards. Remark that, although the one spelling out ab from node I consists of an edge, the path spelling out ab from node B consists of two edges. Anyway, the active point finally arrives in the middle of edge (I, bab, F) while spelling out ab from the bottom node.

3.3 Edge Merging

(The above story still continues here.) Since the active point cannot move with spelling out c from the current location, it seems necessary to create a new node and a new edge labeled with c over there. However, the fact is that letter b is always preceded by letter a in string $ababc$ and that node 1 which corresponds to ab obviously has an edge labeled with c. That is why edge (I, bab, F) is *merged* into node 1 with label b, that is, it becomes $(I, b, 1)$. After that, the active point again moves backwards to I and arrives at B via the suffix link of I. It then stops just on node I spelling out b. Creating a new edge (I, c, F), the active point moves to node B via the suffix link and then moves and stops on node I while spelling out c.

As seen above, thanks to the bottom node, we can obtain the following lemma similarly in [Ukk95].

Lemma 1 *For any string w and any i $(1 \le i \le |w|)$, CDAWG$(w[1:i-1])$ always has the location on which the active point of phase $w[1:i]$ stops.*

The above lemma holds in case of a set of strings, as well.

3.4 Node Separation

The completely opposite thing to edge merging above mentioned, node *separation* can also happen, as seen at phase $ababcb$ in Fig. 5. Recall that the active point was on node I at phase $ababc$. Then since the letter b follows string $ababc$, the active point moves to node 1. Note that it has arrived at node 1 along the edge labeled by b which does not compose the longest path from node I to node 1. Then node 1 is separated into two, that is, a new node 2 is created with the same outgoing edges as those of node 1, and edge $(I, b, 1)$ becomes $(I, b, 2)$. The reason of the above is that letter b is not always preceded by letter a in string $ababcb$, though it was in string $ababc$. If node 1 had incoming edges composing shorter paths between node I and 1 than the path which contains the edge the active point traveled, the last edges in all the paths would be also redirected to node 2.

3.5 Update of Edges Entering to Final Node

Given a set $S = \{w_1, \ldots, w_k\}$, a label of any edge in *CDAWG*(S) is implemented with a triple of integers (h, i, j) such that the label corresponds to $w_h[i:j]$, where $1 \le h \le k$. Let us hereafter call i and j *starting position* and *ending position*, respectively. We make the ending position of every edge which enters to the final node refer to the integer e in the final node. By increasing e each time the CDAWG is extended with a new letter, we obtain the constant time update of all the edges entering to the final node.

As a result of the above discussion, the following theorem holds.

Theorem 1 (Inenaga et al., [IHS$^+$01a]) *For a fixed alphabet, the CDAWG for a set S of strings can be directly constructed on-line, in linear time and space with respect to $\|S\|+|S|$.*

4 Algorithm to Construct the CDAWG for a Trie

We are now ready to show our main algorithm that constructs CDAWGs for tries. First we note that the CDAWG for *Trie*(S) is the same as the CDAWG of S for any set S of string, except only one thing. While the label of an edge in *CDAWG*(S) is implemented by a triple of integers (h, i, j) representing the starting position i and ending position j of the label in the h-th string in S, that in the CDAWG for *Trie*(S) refers to a pair of nodes in *Trie*(S), between of which there is the string corresponding to the label.

The basic action of the algorithm for *Trie*(S) is to update the CDAWG incrementally, synchronized with the depth-first traversal of *Trie*(S). The key idea to achieve the linear time construction is as follows.

(1) Trace the *advanced point* q in the CDAWG so that the path from the root node to q coincides with the path from the root node to node v, where v is the node currently visited in the trie.

(2) Create a new node in the CDAWG where the advanced point q is, before stepping into the first branch at each branching node in the trie.

We will explain the detail in the sequel. Suppose that, after having traveled nodes with scanning $\alpha \in \Sigma^*$ in *Trie*(S), the algorithm encounters a node v having k (≥ 2) branches in *Trie*(S). Moreover suppose that it then chooses an edge with which a path spelling out β and ending at a leaf node begins. After updating the CDAWG with string $\alpha\beta$, the algorithm has to update it with the other strings represented in *Trie*(S). Notice that the current CDAWG already has the path representing α from the initial node, which corresponds to prefixes of at least k strings in S. Thus the algorithm has only to restart updating the CDAWG from the location to which α corresponds and to continue traversing *Trie*(S) from the node v. For that purpose, we trace the *advanced point* q mentioned in (1) above.

Let us now clarify the aim of (2). The aim is to make the advanced point q be an *explicit* node whenever the algorithm encounters a branching node in *Trie*(S). That is, the reference pair of q should then become of the form (s, ε) for some node s. What is the matter if the advanced point q is not explicit before stepping into the first branch? Assume that the advanced point q was referred as (u, γ) with some node u and string $\gamma \neq \varepsilon$ when the algorithm encountered the node v corresponding to α in *Trie*(S). After finishing updating the CDAWG with $\alpha\beta$, the algorithm focuses back on v and $q=(u,\gamma)$. The matter is that the reference (u,γ) might not be *canonical* any longer: the path spelling out γ may contain extra nodes. Namely, the path spelling out γ may have been split while the algorithm updated the CDAWG with string β. A concrete example is shown in Fig. 6.

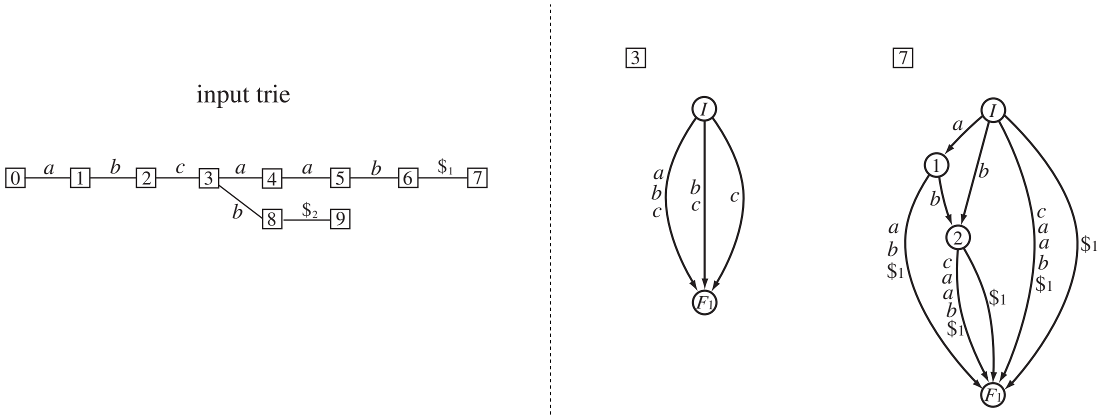

Figure 6: *Trie*(S) for $S = \{abcaab\$_1, abcb\$_2\}$ is shown left. When the algorithm focuses on node 3 in *Trie*(S), it needs to memorize the location in the CDAWG corresponding to abc. Since there is no node but F_1 at the location, it is memorized by a reference pair (I, abc). After having visited node 7 in *Trie*(S), the algorithm updates the CDAWG from (I, abc), and with node 3 in the trie. However, since the path spelling abc dose not consist of an edge any more, the algorithm has to find the nearest node from the location the path ends on, that is, node 2. We have to avoid this, because traversing the path spelling abc in the CDAWG just deserves traversing *Trie*(S) from node 0 to 3.

If the algorithm scans such extra nodes, its time complexity can become quadratic with respect to the number of nodes in *Trie*(S). In order to avoid this matter, the algorithm creates a new node s so that the active point is guaranteed to be on an explicit node. However, the algorithm dose not merge any other edges because at the moment it is unknown how many edges should be merged into the new node s. Of course, if $\gamma=\varepsilon$, there is no need to create any new node.

The algorithm is described as follows.

main routine
  current_node := 0; /* *the root node in the trie* */
  active_point := (I, ε); /* *the initial node in the CDAWG* */
  advanced_point := (I, ε); /* *the initial node in the CDAWG* */
  traverse-and-update(*current_node*, *active_point*, *advanced_point*);

procedure *traverse-and-update*(*current_node*, *active_point*, *advanced_point*)
  Let *label_set* be the set of labels of the outgoing edges of *current_node*;
  if $|$*label_set*$| = 0$ **then return**;
  else if $|$*label_set*$| \geq 2$ **then** *create-node*(*advanced_point*);
  for each $c \in$ *label_set* **do**
    new_active_point := *update-CDAWG*(c, *active_point*);
    Let *new_advanced_point* be the location where *active_point* advances with c;
    Let v be the node to which the edge labeled c points;

traverse-and-update(v, *new_active_point*, *new_advanced_point*);

The variable *current_node* indicates the node that the algorithm currently focuses on in *Trie*(S). The variable *advanced_point* is of the form of a reference pair (u, β), where u is the parent node nearest to *advanced_point*. As mentioned above, the string β is actually implemented by a pair of nodes in *Trie*(S).

In the procedure *traverse-and-update*, the function *update-CDAWG* updates the CDAWG with a letter c. *update-CDAWG* is the same as the one for the construction of the CDAWG for a set of strings [IHS+01a, IHS+01b], excepting that *update-CDAWG* creates a new edge stemming from the node latest created by function *create-node*.

An example of the construction of the CDAWG for a trie is shown in Fig. 7.

Finally, we have the following theorem.

Theorem 2 *The proposed algorithm constructs the CDAWG for a trie in linear time and space with respect to the number of nodes in the trie.*

Proof. We first explain that the modification of the function *update-CDAWG* and the function *create-node* itself do not affect the linearity of the algorithm.

Suppose that an input trie has n nodes. It is clear that the number of nodes visited by *advanced_point* in the CDAWG is at most n. Hence it takes $O(n)$ time to calculate *advanced_point* all through the construction. Furthermore suppose that m nodes in *Trie*(S) are branching. It is clear that $m < n$, because any trie has at least one leaf node. Therefore, function *create-node* creates at most m nodes in the CDAWG, and it implies that the time complexity of *create-node* is $O(m)$. This implies the modification, creating new edges due to the nodes made by function *create-node*, takes $O(m)$ time as well.

We from now on verify the overall linearity of the proposed algorithm. The matter we have to clarify is the upper bound of the number of nodes *active_point* visits throughout the construction. Assume that a node v in the trie has k branches and there is a path spelling α between the root and v. When *current_node* arrives at node v in the trie for the first time, function *create-node* creates a new node u where *advanced_point* is in the CDAWG. Then *active_point* may traverse at most $k|\alpha|$ nodes from p to the initial node via suffix links until finding the location it can stop on. However, $k \leq |\Sigma|$. Therefore, for a trie with n nodes, the number of nodes *active_point* visits throughout the construction is $O(|\Sigma|n)$. Thus, if Σ is a fixed alphabet, the proposed algorithm constructs the CDAWG for a trie in $O(n)$ time and space. □

5 Conclusion

We gave an algorithm for constructing the CDAWG for a trie in linear time and space with respect to the number of nodes in the trie. The truth is, with a slight modification, the proposed algorithm can be adopted to construct the suffix tree and the DAWG for a trie. When input strings are given in the form of a trie, the proposed algorithm constructs the CDAWG for the strings faster than the one presented in [IHS+01a] directly does from a set of the strings, especially when the strings have many common prefixes. As the space complexity of CDAWGs is bounded strictly

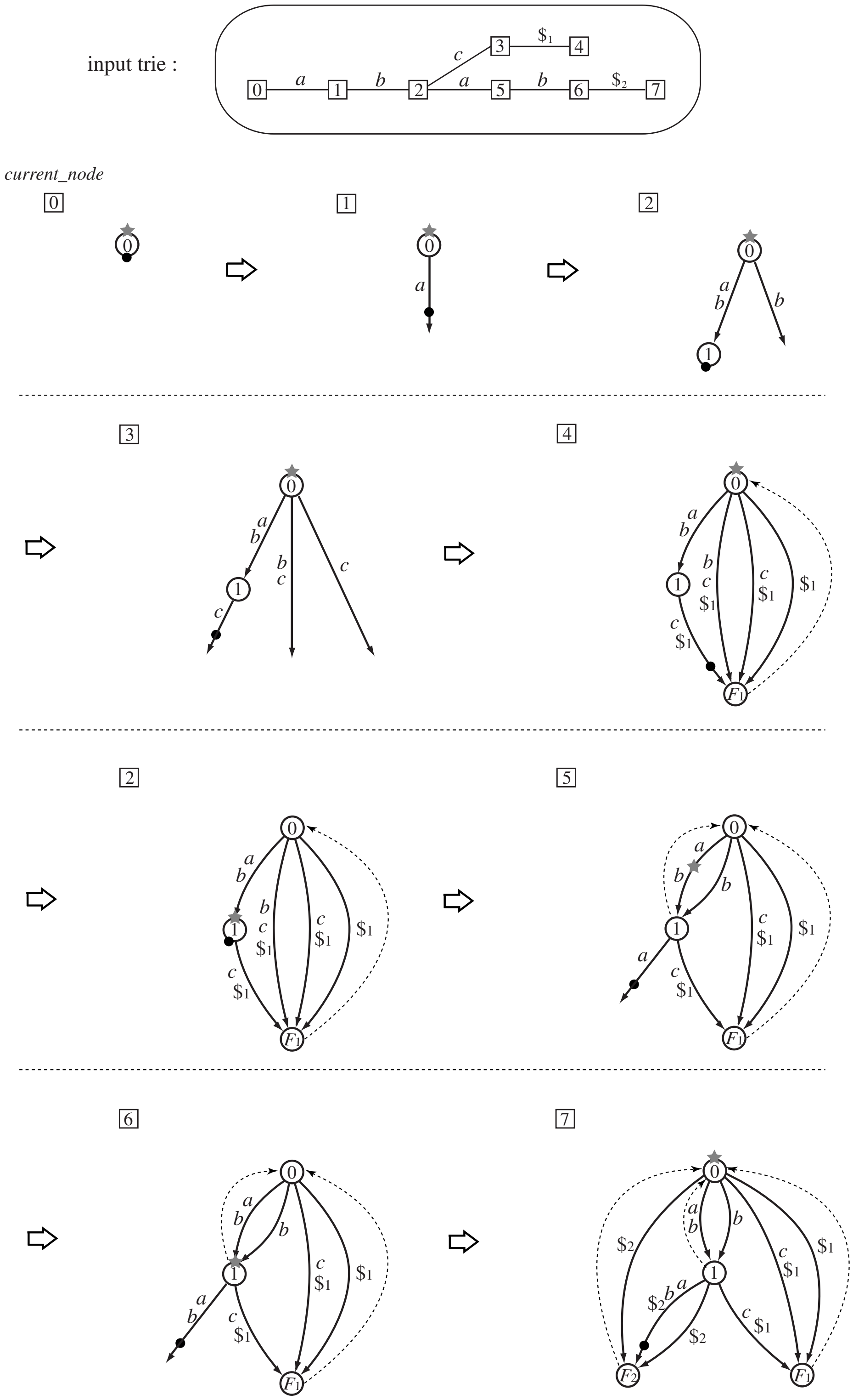

Figure 7: Construction of the CDAWG for *Trie*(S), where $S = \{abc\$_1, abab\$_2\}$. The gray starred point represents *active_point*, and the black dotted point represents *advanced_point*. For simplicity, the bottom node is omitted. As node 2 in the trie is branching, a new node 1 is created in the CDAWG when *current_node* arrives at node 2 for the first time. After *current_node* visits node 4, the algorithm updates the CDAWG with *current_node* $= 2$ and *advanced_point* $= 1$.

lower than that of suffix trees, the algorithm presented in this paper also allows to save memory space.

References

[BBH+85] Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[BBH+87] Anselm Blumer, Janet Blumer, David Haussler, Ross McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987. Preliminary version in: STOC'84.

[Bre98] Dany Breslauer. The suffix tree of a tree and minimizing sequential transducers. *Theoretical Computer Science*, 191:131–144, 1998.

[CV97] Maxime Crochemore and Renaud 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.

[IHS+01a] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs (to appear). In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, July 2001.

[IHS+01b] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, and Setsuo Arikawa. On-line construction of compact directed acyclic word graphs. Technical Report DOI-TR-CS-183, Department of Informatics, Kyushu University, January 2001. (To appear).

[Kos89] S. Rao Kosaraju. Fast pattern matching in trees. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 178–183, 1989.

[McC76] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, October 1973.

Genome and Stringology

Laurent Mouchard

ABISS – Dept of Vegetal Physiology
Université de Rouen
76821 Mont Saint Aignan Cedex
France
and
Dept of Computer Science
King's College London
Strand, London, WC2R 2LS
England

e-mail: Laurent.Mouchard@univ-rouen.fr

Abstract. The sequence of the human genome is a 3.2 billion letter string built upon a four letter alphabet. In this string, bioinformics researcher are trying to find the complete set of genes that produce proteins. Genes represent 3 to 5 of the length of the human genome, most of this genome being non-coding sequences, including highly repetitive regions, non-activated copies of genes, ...

The public databases containing biological sequences that world-wide scientists are feeding everyday have grown exponentially for the last 15 years and the flood of genomic data, produced by a large number of genomics effort, has begun to affect the way we can deal with information. We are producing more information than we can handle.

The interaction between biologists and computer scientists can greatly benefit to computer scientists (and stringologists) by proposing new problems, news applications to problems that have been solved previously in a different context and asking for additional extensions.

The first part of my talk will be a presentation of a wide range of problems where stringology and molecular biology (more specifically genomics research) meet, from traditional pattern matching problems to shortest common superstring, repeat detection, motif modelisation, comparison of multiple strings, infering 3D structure from a 1D sequence and so on. I will also give several examples of biology-stringology-graph theory problems, from the assembly of the human genome to the Sequencing By Hybridation.

In the second part of my talk, I will focus on a couple of these problems, that is: - sequence comparison - I will discuss the traditional solutions that have been proposed and used so far, their implementation on sequential and parallel computers or dedicated cards and will discuss new problems arising in this restricted field. - repeat detection, where several kinds of repeats are studied: tandem and non-tandem repeats, exact or approximative to name a few and will discuss new open problems we encounter recently.

Bioinformatics: tools for analysis of biological sequences

Jan Pačes

ABISS – Dept of Vegetal Physiology
Université de Rouen
76821 Mont Saint Aignan Cedex
France
and
Dept of Computer Science
King's College London
Strand, London, WC2R 2LS
England

e-mail: `Laurent.Mouchard@univ-rouen.fr`

abbreviations: DNA - deoxyribonucleic acid; bp - base pair; contig - a long region of DNA assembled from shorter DNA sequences

1 Genomics and Bioinformatics

Information that directs all processes in living cells is stored in sequences of nucleotides in deoxyribonucleic acid (DNA). Contemporary methods of determining the nucleotide sequences, the so-called sequencing of DNA, are so effective that sequencing of whole genomes became feasible.

The field of Genomics is aimed at complex analysis of genomes based on our knowledge of the nucleotide sequences in DNA. Complete structures of several tenths of genomes have been determined so far (see http://kegg.genome.ad.jp/kegg/catalog/ /org_list.html or http://www.tigr.org/tdb/mdb). Most of them are bacterial genomes. These genomes usually consist from one chromosome sometimes accompanied by one or several plasmids. Bacterial chromosomes and plasmids are circular molecules of DNA. The number of nucleotides in prokaryotic (e.g. bacterial) genomes ranges from a fraction of million to several millions. Several genomes of higher eukaryotic organisms were also sequenced. They are the genomes of the yeast *Saccharomyces cerevisiae* (12 Mbp), the worm *Caenorhabditis elegans* (97 Mbp), the fruit fly *Drosophila melanogaster* (137 Mbp) and the plant *Arabidopsis thaliana* (116 Mbp). In 2001 nearly complete nucleotide sequence of the human DNA was announced. The complete human genome consists of more than three billion nucleotides. Several genome projects are completed every months. Most of them are bacterial genome projects, but genomes of several higher organisms (e.g. mouse or chimp) are getting close to completion.

Accumulation of the vast number of nucleotide sequences calls for a robust computer analysis. Bioinformatics is a new field devoted to analysis of long strings of nucleotide sequences generated in genome projects. Analysis of amino acid sequences of proteins encoded in the genomes usually follows the analysis of DNA and is an important part of bioinformatics.

To obtain the complete genome sequence it is necessary to assemble stepwise long nucleotide strings from the shorter nucleotide sequences usually generated from individual clones. In a typical example a sequence of several hundred nucleotides is determined in one sequence run. From these partial sequences longer and longer strings (the so-called contigs) are assembled until the complete genome sequence is obtained.

The long contigs, in the ideal case the complete genome, are subjected to further computer analysis. We try to identify all genes present in the nucleotide sequence, to elucidate their structure (e.g. exon-intron organization), find elements regulating gene expression (e.g. promotors, enhancers, transcription terminators) and to identify other important DNA features. Genes are translated into the sequence of amino acids of the corresponding proteins. From these amino acid sequences basic features of the proteins are derived. This may for instance be the protein's secondary structure. Usually the overall DNA characteristics such as its base composition is also described.

After this basic DNA characterization nucleotide and amino acid sequences are often compared with the entries in international databases. Contemporary databases are very large. For instance the EMBL database of nucleotide sequences contains more then ten billion nucleotides of many genes and genomes. This number grows exponentially.

From the similarities found by this search we can often ascribe functions to individual genes and the corresponding proteins. The ultimate goal is to describe the complete metabolism of the organism. An important result of these comparisons are evolutionary relationships among organisms. It is now possible to describe on the molecule level individual taxons.

2 Features of Biological Sequences

Nucleotide and amino acid sequences have several special features that have to be taken into account when performing computer analysis. These special features are connected with the biochemical and biological function of genes and proteins. For instance, variations in nucleotide sequences performing the same function (e.g. promoters) make the analysis difficult.

2.1 DNA

DNA is the polymer molecule in which genetic information of organisms is stored. DNA consists of four basic components, the so-called nucleotides. Each nucleotide consists of a sugar deoxyribose, a residue of phosphate and one of the four nitrogenous bases. These basis are adenine (A), guanine (G), cytosine (C) and thymine (T). Nucleotides are connected by sugar-phosphate bonds into long strings. DNA is composed from two strings that run antiparallel in the well known double helix. The two strings (strands) are complementary. This means that A in one strand pairs through hydrogen bonds with T in the other strand and G pairs with C. Genetic information stored in the sequence of A, C, G and T reads in one direction only. Because the two strands are antiparallel genetic information reads in the two strands in opposite direction. For instance the sequence ATTGCA in one strand reads TGCAAT in the complementary strand.

In databases and for computer analysis the DNA sequences are stored in the single-letter code. Because of occasional ambiguities in sequence analysis additional letters are used to facilitate nucleotide analysis (Tab. 1).

Table 1: Nucleotide code. Compl. stands for complementary nucleotide.

name	code	nucleotide	compl.
Adenine	A	A	T
Cytosine	C	C	G
Guanine	G	G	C
Thymine	T	T	A
Uracil	U	U	A

code	nucleotide	compl.
M	A/C	K
R	A/G	Y
W	A/T	S
S	C/G	W
Y	C/T	R
K	G/T	M
V	A/C/G	B
H	A/C/T	D
D	A/G/T	H
B	C/G/T	V
N	A/C/G/T	N
-	space	-

2.2 Translation of DNA to Proteins

Regions of DNA to which most attention is devoted are genes. In prokaryotic cells genes usually consist of uninterrupted nucleotide sequence. In contrast, genetic information of eukaryotic organisms is organized in a more complex fashion. Regions encoding amino acids (exons) are interspaced by long non-coding sequences (introns). Thus only two to three percent of human DNA encodes proteins.

A sequence of three nucleotides, the so-called triplet or codon, determine which amino acid is incorporated in the corresponding protein. Proteins consist of 20 types of two amino acids. With the exception of two amino acids all are encoded by more then one codon. This implies that although translation of DNA to protein is unique it is impossible to unequivocally derive nucleotide sequence from the amino acid sequence. In addition, a region of DNA can encode six different proteins.

Genetic code is in Table 2.

2.3 Proteins

Proteins are the functional molecules operating in cells. Basic building blocks of proteins are amino acids. Proteins consists of 20 types of amino acids. For amino acids a one letter code is used, although the older three letter code can be occasionally also found in biochemical literature (Tab. 3).

From the biochemical point of view it is important that unlike nucleotides amino acids generally are much more chemically different. However, some of them are so similar that they can replace each other in functional proteins. For example leucine,

Table 2: Genetic code

1	2	A	G	C	T	3
1	A	Lys	Arg	Thr	Ile	A
		Lys	Arg	Thr	Met	G
		Asn	Ser	Thr	Ile	C
		Asn	Ser	Thr	Ile	T
	G	Glu	Gly	Ala	Val	A
		Glu	Gly	Ala	Val	G
		Asp	Gly	Ala	Val	C
		Asp	Gly	Ala	Val	T
	C	Gln	Arg	Pro	Leu	A
		Gln	Arg	Pro	Leu	G
		His	Arg	Pro	Leu	C
		His	Arg	Pro	Leu	T
	T	*	*	Ser	Leu	A
		*	Trp	Ser	Leu	G
		Tyr	Cys	Ser	Phe	C
		Tyr	Cys	Ser	Phe	T

isoleucine and valine are often found in the same position in functionally identical proteins isolated from different organisms.

Table 3: Amino acid code

1-code	3-code	amino acid
A	Ala	alanine
C	Cys	cysteine
D	Asp	asparagic acid
E	Glu	glutamic acid
F	Phe	phenylalanine
G	Gly	glycine
H	His	histidine
I	Ile	isoleucine
K	Lys	lysine
L	Leu	leucine
M	Met	methionine
N	Asn	asparagine
P	Pro	proline
Q	Gln	glutamine
R	Arg	arginine
S	Ser	serine
T	Thr	threonine
V	Val	valine
W	Trp	tryptophan
Y	Tyr	tyrosine
B	Asx	aspartic acid or asparagine
Z	Glx	glutamic acid or glutamine
X	Xxx	any amino acid
*	—	stop

3 Main types of analyses

The most frequent tasks of DNA and protein analysis are:

- Assembly: to assemble contigs from short (several hundred nucleotide long) sequences.

- Gene prediction: to identify genes in DNA.
- Pattern search: to find regions composed of typical short sequences.
- Pairwise alignment: to find in a database similar or homologous sequence.
- Multiple alignment: to assess relationship of several sequences.

4 Comparison of Biological Sequences

The basic step in analyzing a determined nucleotide sequence is its comparison with the sequences already deposited in databases. We are looking for related and/or similar sequences. The presumption for this search is that in evolution nucleotide substitutions, and less frequently nucleotide deletions and insertions, are accumulated. Thus similarity of genes and proteins can be traced in more or less distantly related organisms. Accumulation of these changes is not random. It is more frequent in the DNA regions encoding those part of proteins that are not fundamentally important for the protein's function.

An algorithm for pairwise comparison of amino acid sequences was described for the first time by Needleman & Wunsch in [1] and is known as the global Needleman-Wunsch algorithm. Because in most cases two sequences are more similar in certain regions and less similar in other in other regions, the Needleman-Wunsch algorithm was improved for evaluation of local similarities. This is called local Smith-Waterman alignment [2] [3] [4].

The basic principle of the similarity search is known as the pairwise alignment and is based on comparison of sequences pair by pair and on search for the best alignment with highest similarity. Score of bonuses and penalizations for matches, mismatches, deletions etc. are calculated for all possible pairs. Scores of all alignments are then calculated as the sum of scores of all pairs in the alignment. The best similarity it then ascribed to the alignment with the highest score. This is known as the Smith-Waterman score and is usually used as basic characteristics of the alignment.

4.1 Scoring Matrix

When comparing two nucleotide sequences, it is not necessary to evaluate similarity of individual nucleotides. It is sufficient to consider identities. Fundamentally different are comparisons of amino acid sequences. When analyzing evolutionarily related proteins it was discovered that for enzyme activities general biochemical properties of individual amino acids are very important. For many comparisons it is useful to group amino acids according to their chemical properties such as hydrophobicity, charge, size, polarity etc. Substitution of amino acids belonging to one such group may be penalized less compared to substitutions of unrelated amino acids.

However, it is also important to take into account genetic (evolutionary) relatedness of individual amino acids. For instance, tryptophane is encoded by the TGG codon. One mutation leads to codons for glycine (GGG), serine (TCG) and leucine (TTG), two codons for cysteine (TGT, TGC), arginine (CGG, AGG) and two stop codons (TGA, TAG). Conversion of tryptophane to arginine is therefore more likely than conversion to glycine in spite of the fact that tryptophane and arginine are chemically very different: tryptophane is hydrophobic aromatic amino acid and arginine is hydrophilic polar positively charged amino acid.

These considerations are taken into account in the so-called scoring matrix, which is basically evaluation of a replacement of one amino acid by another amino acid. Today we use two types of scoring matrix: PAM and BLOSUM. PAM and BLOSUM differ by the calculation method and they give similar results. Table 4 shows a part of the scoring matrix concerning tryptophane (W).

Table 4: Scoring matrix for tryptophane (W) and its change to selected amino acids (R,N,D,C).

W	R	N	D	C	W
PAM 50	-1	-7	-12	-13	13
PAM 100	1	-5	-9	-9	12
PAM 250	2	-4	-7	-8	17
BLOSUM 100	-7	-8	-10	-7	17
BLOSUM 62	-3	-4	-4	-2	11
BLOSUM 30	0	-7	-4	-2	20

5 Often Used Programs

Demands for memory and computational time grows with the length of the biological sequence under evaluation and with volume of the database. This is why an important principle is pre-selection of possible hits. This pre-selection is based on observation that two evolutionarily distant sequences usually contain conserved short regions. These conserved regions can served for fishing out evolutionarily related sequences. Most commonly used programs based on this principle are FASTA and BLAST, which are designed to compare both nucleotide and amino acid sequences. Moreover, these two programs deal with a basic feature of genetic information, i.e. conversion of the language of the nucleotide sequences in DNA into the language of amino acids sequences in proteins. This is done in the six possible reading frames.

5.1 FASTA

The FASTA program (FAST Algorithm) [5] was one of the first freely available programs of bioinformatics. FASTA first prepares list of "words", i.e. very short portions of the sequence in question. This list is then compared with the entries in the database. If several words match an entry in the database in the right order and close-by the sequence is selected for the complete Smith-Waterman alignment.

There are several variants of the FASTA program, according to the type of the sequence and database used. The main FASTA program is used for searching nucleotide sequences in DNA databases or amino acid sequences in protein databases. FASTX/FASTY are used for comparison od DNA against proteins. TFASTX/TFASTY can be used to look for proteins in DNA databases.

An important parameter of the search is "expectancy" (E). E is proportional to the probability with which the same degree of similarity can be found in a random sequence of the same length. Because Smith-Waterman score depends on the sequence length it was normalized to the unit of length (so-called "bit score")

Next is an example of the FASTA search.

```
 FASTA searches a protein or DNA sequence data bank
 version 3.3t09 May 18, 2001

428405286 residues in 67627 sequences
 statistics extrapolated from 60000 to 69642 sequences
  Expectation_n fit: rho(ln(x))= 13.2503+/-0.000153; mu= -20.9826+/- 0.010
 mean_var=452.3589+/-70.442, 0's: 3 Z-trim: 470  B-trim: 62 in 1/86
 Lambda= 0.0603

FASTA (3.39 May 2001) function [optimized, +5/-4 matrix (5:-4)] ktup: 6
 join: 73, opt: 58, gap-pen: -16/ -4, width:  16
 Scan time: 16.480
The best scores are:                                     opt bits E(69642)
EM_INV:DM19269 U19269 Drosophila melanogaste (4976)  [f] 736   80 6.2e-13
EM_INV:AC007185 AC007185 Drosophila melanoga (77732) [r] 723   80 7.8e-13
EM_INV:AF139019 AF139019 Cepaea nemoralis mi ( 624)  [r] 452   55 2.6e-05
...

>>EM_INV:DM19269 U19269 Drosophila melanogaster Dachshun  (4976 nt)
 initn: 870 init1: 692 opt: 736  Z-score: 352.9  bits: 80.3 E(): 6.2e-13
 59.627% identity (62.036% ungapped) in 644 nt overlap (242-877:976-1602)

             220       230       240       250       260       270
gi      CAGTCACCTCTCCTGGTGGCGGCGGCGGCGGCAGCGGAGGCGGCGGTGGCAGCGGCGGCA
                                     ::::: :: :: ::     :::: ::  ::
EM_INV TCCGGTGAGCTCCCTCAACCACTCCATGATGCAGCAGATGCAGC---AACAGCAGCAACA
         950       960       970       980          990      1000

             280       290       300       310       320       330
gi      ACGGAGGCGGCGGCGGGAGCAACTGCAACCCCAGCCTGGCGGCCGGGAGCAGCGGCGGCG
       :: :  :: ::  : : :::: : ::::: ::: :    : :::    ::  :   :
EM_INV ACAGCAGCAGCAACAGCAGCAGCAGCAACACCATCAGCTCAGCCCCCCGCCACATGGAAT
           1010      1020      1030      1040      1050      1060

             340          350       360       370       380
gi      GCGTTAGCG---CTGGCGGCGGCGGCGCCTCCAGCACCCCCATCACCGCGAGCACCGGCA
       ::  : : :   : : : :: :  : :::: : :: ::   ::  ::     ::  : :
EM_INV GCCATCGGGCAACGGACTGCCGACGGGCCTACCGC-CCAGAATGCCC-----CATGGACT
           1070      1080      1090       1100           1110

...
```

5.2 BLAST

BLAST (Basic Local Alignment Search Tool) [6] [7] [8] was developed in 1990. BLAST first maps the database for presence of various strings and lists them. Then, similarly to FASTA, it creates a list of "words" from the searched sequence. These words are then compared to the list of strings. This leads to selection of the best hits which are then extended from both sides. The BLAST program has several variants similarly to FASTA.

6 Multiple Alignment

A very important task in evaluating biological sequences is comparison of more then two sequences at the same time. This is called multiple alignment. Multiple alignment helps to identify biologically important parts of genes. In addition it enables to estimate evolutionary distances among biological sequences, to formulate consensus sequences and parental sequences. Especially important are consensus sequences because they can serve in identification of additional, often more distant members of the gene family in question.

The most common program for the multiple alignments is CLUSTALW ([9] [10] [11] [12]). CLUSTALW compares by the Needleman-Wunsch algorithm all sequences in the query and it selects the most similar pair. From these two sequences a consensus sequence is generated and it is used for aligning another sequence. With this stepwise mechanism whole family of sequences are compared. The relatedness of individual members of a sequence family can be assessed..

An example of the CLUSTALW search follows.

```
Hs-U3   LDALSRECCVTAGGRDGTVRVWKI----PEESQLVFYGH--------------------------
Mm-U3   LDALSRECCVTAGGRDGTVRVWKI----PEESQLVFYGH--------------------------
Xl-U3   LDSLSRERCVTVGGRDGTMRIWKI----AEETQLVFSGH--------------------------
At-U3a  IDALRKERALTVG-RDRTMLYHKV----PESTRMIYRAP--------------------------
At-U3b  IDALGRERVLSVG-RDRTMQLYKVGIVVPESTRLIYRAS--------------------------
Dm-U3   IDALSRERAITAGGSDCSLRIWKI----TEESQLIYNGH--------------------------
Sp-U3   VDALARERCVSVGGRDRTSRLWKI----VEESQLVFRSGGTSMKAT-----AGYM----------
Nc-U3   IDALAGERCVSVGARDRTARYWKV----PEESQLVFRGGVSEKKSHKNRDQAVNH----------
Ce-U3   IGVLSKQRVATVGGRDRSARLWKV----EDESQLMFSGLQN------------------------
Sc-U3   ISALAMERCVTVGARDRTAMLWKI----PDETRLTFRGGDEPQKLLRRWMKENAKEGEDGEVKYPD
        :. *  :   :.*  * :    *:     :.::: : .

Hs-U3   ---------QGSIDCIHLINEEHMVSGADDGSVALWGLSKKRPLALQREAHGLRGE----------
Mm-U3   ---------QGSIDCIHLINEEHMVSGADDGSVALWGLSKKRPLALQREAHGLHGE----------
Xl-U3   ---------EGSIDCVRLINEEHIVTGADDGSLALWTVGKKKPLTQMKCAHGSYGE----------
At-U3a  ---------ASSLESCCFISDNEYLSGSDNGTVALWGMLKKKPVFVFKNAHQDIPDGITTNGILEN
At-U3b  ---------ESNFECCCFVNSDEFLSGSDNGSIALWSILKKKPVFIVNNAHHVIAD----------
Dm-U3   ---------KDSIECVKYINDEHFVSGGMDGAIGLWSALKKKPICTTQLAHGVGEN----------
Sp-U3   ---------EGSVDCVAMIDEDHFVTGSDNGVIALWSVQRKKPLFTYPLAHGLDPILAPGRHSAET
Nc-U3   ---------DGTMDQVAMIDDELFVTGDSAGTLSLWGINRKKALFTQPCAHGIDPPLKPTEVSADA
Ce-U3   ---------CVSLDCVAMINEEHFATGSADGSIALWSFWKKRALHVRKQAHGTQNG----------
Sc-U3   ESEAPLFFCEGSIDVVSMVDDFHFITGSDNGNICLWSLAKKKPIFTERIAHGILPEPSFNDISGET
                   ..:    :..    :*   * : **   :*:.:     **
```

References

[1] Wunsch CD Needleman SB. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3), 1970.

[2] Waterman MS Smith TF. Identification of common molecular subsequences. *J Mol Biol*, 147(1), 1981.

[3] Taylor P. A fast homology program for aligning biological sequences. *Nucleic Acids Res*, 12(1 Pt 2), 1984.

[4] Waterman MS. Efficient sequence alignment algorithms. *J Theor Biol*, 108(3), 1984.

[5] Lipman DJ Pearson WR. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8), 1988.

[6] Altschul SF et al. Basic local alignment search tool. *J Mol Biol*, 215(3), 1990.

[7] States DJ Gish W. Identification of protein coding regions by database similarity search. *Nat Genet*, 3(3), 1993.

[8] Altschul SF et al. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17), 1997.

[9] Sharp PM Higgins DG. Clustal: a package for performing multiple sequence alignment on a microcomputer. *Gene*, 73(1), 1989.

[10] Fuchs R Higgins DG, Bleasby AJ. Clustal v: improved software for multiple sequence alignment. *Comput Appl Biosci*, 8(2), 1992.

[11] Higgins DG. Clustal v: multiple alignment of dna and protein sequences. *Methods Mol Biol*, 25, 1994.

[12] Gibson TJ Thompson JD, Higgins DG. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22(22), 1995.

Acknowledgements: This work was supported by Center of Integrated Genomics and by grant GA301/99/M023 of the Grant Agency of the Czech Republic.