Department of Computer Science and Engineering

Towards Faster String Matching

Hannu Peltola

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 78/2013

Towards Faster String Matching

Hannu Peltola

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 (Konemiehentie 2, Espoo) of the school on 10th May 2013 at 12 noon.

Aalto University
School of Science
Department of Computer Science and Engineering

Supervising professor
Professor Jorma Tarhio

Thesis advisor
Professor Jorma Tarhio

Preliminary examiners
Assistant Professor Simone Faro, Università di Catania, Italy
Ph.D. Kimmo Fredriksson, University of Eastern Finland, Finland

Opponent
Ph.D. Juha Kärkkäinen, University of Helsinki, Finland

Aalto University publication series
DOCTORAL DISSERTATIONS 78/2013

ISBN 978-952-60-5156-7 (printed)
ISBN 978-952-60-5157-4 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-5157-4

Unigrafia Oy
Helsinki 2013

Finland

NORDIC ECOLABEL
441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Hannu Peltola

Name of the doctoral dissertation
Towards Faster String Matching

Publisher Aalto University School of Science

Unit Department of Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 78/2013

Field of research Software Technology

Manuscript submitted 11 December 2012 **Date of the defence** 10 May 2013

Permission to publish granted (date) 29 January 2013 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

Exact string matching is a much studied and popular problem. The task is to find the occurrences of a string pattern in a long text. We introduce several new algorithms for online exact string matching and study their experimental performance and compare them with some earlier algorithms. Moreover, we give examples of anomalies caused by repetitive texts, and present algorithms for that kind of data. Specialized algorithms for long patterns (longer than about 50 characters) are also studied. In addition, we discuss the principles of fair testing of string matching algorithms.

Among the designed algorithms there are novel ones and variations of previous algorithms. Most algorithms are based on bit-parallelism, which is one of the most effective techniques adopted increasingly during the last few years. For example classical exact string matching algorithms were significantly slower than the best ones of the new algorithms.

Exact string matching is so common task that even a small improvement is useful in practice. It turned out that the uncomplicated algorithms are often the fastest in practice, especially for short patterns. Reducing the number of tests in algorithms often accelerates search. Surprisingly, the number of examined text characters is not always correlated with practical search speed among various exact string matching algorithms. Typically, there is clear dependence on the number of examined text characters and performance only among closely related algorithms. However, no exact string matching algorithm is the best one for all alphabets and pattern lengths on all kinds of computer hardware.

Keywords exact string matching, q-gram, bit-parallelism, BNDM, Horspool's algorithm

ISBN (printed) 978-952-60-5156-7 **ISBN (pdf)** 978-952-60-5157-4

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2013

Pages 220 **urn** http://urn.fi/URN:ISBN:978-952-60-5157-4

Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Hannu Peltola

Väitöskirjan nimi
Nopeampaa merkkijononhakua

Julkaisija Aalto-yliopisto Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 78/2013

Tutkimusala Ohjelmistotekniikka

Käsikirjoituksen pvm 11.12.2012 **Väitöspäivä** 10.05.2013

Julkaisuluvan myöntämispäivä 29.01.2013 **Kieli** Englanti

☐ **Monografia** ☒ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Tarkka merkkijononhaku on pitkään tutkittu ja edelleen suosittu tutkimusongelma. Tehtävänä on löytää tekstistä kohdat, joissa esiintyy haettava merkkijono, jota kutsutaan hahmoksi. Tässä työssä esitetään tarkkaan merkkijonohakuun uusia algoritmeja, tutkitaan niiden suorituskykyä käytännössä ja vertaillaan niitä useisiin tunnettuihin kilpaileviin algoritmeihin. Sisällöltään toisteisista aineistoista haettaessa esiintyviä erikoistilanteita analysoidaan ja niihin esitetään algoritmeja. Työssä tarkastellaan myös pitkien – noin 50 merkkiä pidempien – hahmojen etsintää. Lisäksi käsitellään reilun ja tasapuolisen suoritusnopeustestauksen periaatteita sekä mittaustarkkuutta.

Kehitetyistä algoritmeista osa on uusia ja toiset taas muunnelmia aiemmista. Useimmat esitetyt algoritmit hyödyntävät bittirinnakkaisuutta, joka on viime vuosina laajasti omaksuttu tehokas tekniikka. Monet ''klassiset'' algoritmit osoittautuivat usein selvästi hitaammiksi kuin uudet.

Merkkijononhaku on sovelluksissa yleinen ongelma, ja toisinaan tutkittavat tekstit ovat niin suuria, että pienilläkin parannuksilla on käytännöllistä merkitystä. Osoittautui, että yleensä suoraviivaisin menetelmä on nopein; näin erityisesti lyhyillä hahmoilla. Ehdollisia hyppykäskyjä aiheuttavien testien vähentäminen nopeuttaa hakua. Hieman yllättäen tekstistä käsiteltävien merkkien määrän vähentäminen ei läheskään aina tuota nopeampaa algoritmia. Tyypillisesti tämän tyyppinen riippuvuus pätee vain hyvin samanlaisilla algoritmeilla. Mikään merkkijononhakualgoritmi ei ole nopein kaikilla aakkostoilla ja hahmon pituuksilla erilaisilla (tietokone)laitteistoilla.

Avainsanat tarkka merkkijononhaku, q-gram, bittirinnakkaisuus, BNDM, Horspoolin algoritmi

ISBN (painettu) 978-952-60-5156-7 **ISBN (pdf)** 978-952-60-5157-4

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Espoo **Painopaikka** Helsinki **Vuosi** 2013

Sivumäärä 220 **urn** http://urn.fi/URN:ISBN:978-952-60-5157-4

Foreword

Quite often, performing scientific research is similar to trying to hit a moving target, since the scientific goal has to adapt to emerging new problems. Everyone knows that a moving target is hard to hit. This study—like all practical work on continuously evolving computers—was conducted on ever-changing platforms. Work on this thesis has taken long time. During those years I had the privilege to work with and meet numerous great and inspiring persons, whose contribution I would like to acknowledge here. I mention some of them, without diminishing the role of the others.

I'm deeply grateful to the String Research Group and the department of Computer Science and Engineering at Aalto University for providing a great working environment. More specifically, I would like to thank my mentor Prof. Jorma Tarhio for his guidance, supervision, excellent cooperation and support during my PhD work. I am also thankful to all coauthors in publications in this thesis: Branislav Ďurian, Jan Holub, Petri Kalsi, and Leena Salmela. Special thanks to Riku Saikkonen for his knowledgeable answers to my numerous questions.

I want also thank Hans Söderlund for teaching me a lot and giving good example of making research. I am grateful about the courage of Lauri Malmi and Raimo Ruottinen in showing their trust to my work. It gave more strength. I am indebted to Sami Khuri for reading the first draft of this thesis and for helping me polish the text.

I would also like to thank the various anonymous referees for their comments.

My deepest gratitude goes to my wife Aino for believing in and supporting me.

Finally, I am most thankful to Prof. Simone Faro and Ph.D. Kimmo Fredriksson for pre-examining my manuscript and giving insightful comments for it. I would also like to acknowledge Ms. Anya Siddiqi's consulting with English language. Finally, I would like to thank Ph.D. Juha Kärkkäinen for his role as dissertation opponent.

Helsinki, April 16, 2013,

Hannu Peltola

Contents

List of Algorithms

1. Introduction

The amount of information available on the internet has grown enormously, thus making the extraction and location of information more laborious. Compared to the images and videos, searching for textual information is relatively painless. However, search for occurrences of given string is most common task, and therefore the solutions should be efficient. Good solutions were already devised several decades ago. On the other hand, development of hardware offers opportunities and causes needs for better solutions. Therefore new methods for string matching are still being developed.

The string, whose occurrences are searched for is called the *pattern*. It can be a word, a partial word, or even a text fragment. The document, which is examined during the search, is called the *text*. It commonly consists of some natural language, but it can be any kind of binary data in a computer.

The worst case computational complexity of many exact pattern matching algorithms on strings is $\mathcal{O}(mn)$, where m is the length of the pattern and n is the length of the text. Nevertheless, it is typically assumed that the distribution of characters is (discrete) uniform and that characters do not depend on each other. However, an improvement in the worst case complexity does not necessarily imply a better average performance in practice.

The focus in this thesis is on practical search speed. How many megabytes can be processed within a second. All good string pattern matching algorithms are so fast that for many applications the underlying system issues such as I/O management are now performance bottlenecks [HuS91]. Therefore, it is not appropriate to consider the real time clock, and it is better to view the CPU time usage. This work includes tests about practical speed of algorithms. The measurements have been made on several

computers, because the relative performance of algorithms depends on computer hardware. The recent development in branch prediction and speculative execution of processors has improved their performance. We study whether the reduction of the number of conditional branches improves the search speed. For example, when does unrolling help?

Many existing algorithms utilize (hidden) assumptions about the texts. These premises offer several ways to speed up searching. For example, placing a copy of the pattern beyond the end of the text allows an advantageous reformulation of loops. The speed of exact pattern matching algorithms depends heavily on the data and especially on the pattern. Special care will be paid to the careful implementation of algorithms to prevent ruining the performance with unfavorable implementation.

1.1 Problem

Let us next define the problem and terminology precisely. A *string* is a sequence of *characters* over a finite alphabet Σ of c characters. A non-negative integer number 4514799 is a string consisting of seven characters over the alphabet {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Given a string P called *pattern* and a longer string T called *text*, the **exact string matching problem** is to find all occurrences, if any, of the pattern P in the text T [Gus97].

Preprocessing means initializations and other processing that is carried out before the actual search begins. Algorithms that can do preprocessing based on the pattern, but nothing that uses information about the text, are called online algorithms. This work considers only online algorithms.

Nowadays characters are typically presented in computers in one or more bytes. With variable length character encodings, the pattern and the text are treated as a sequence of bytes. Fixed length character encodings consisting of more than one byte are briefly dealt with. So basically we have the byte[1] oriented approach. The UTF-16 character coding can be handled either in bytes or as two byte words. It is assumed that basic elements of characters can be accessed in an identical way and speed. This means that a search in compressed texts is not considered.

[1] In the programming language C byte is defined 'addressable unit of data storage large enough to hold any member of the basic character set of the execution environment'.

All the algorithms discussed here are sequential. On the other hand, so called bit-parallelism (described in subsection 2.5) has been used quite successfully. During the making of this work the "instructionlevel parallelism" has spread so that nowadays it is a common feature in efficient processors. To utilize it extensively, it is essential to try to keep the processor pipeline running. This can be achieved by avoiding conditional branches and trying to keep them as predictable as possible, when they are needed.

The exact string matching problem is often assumed identical to the default output of GNU grep[2] utility. The reporting of line numbers need laborious additional bookkeeping of line changes (e.g. linefeeds): "The Boyer–Moore algorithm cannot be used, as far as we know, for most of the extensions in the previous section; even finding line numbers for all the matches is not trivial and slows the algorithm down considerably" [WuM91]. Apparently, they meant that simultaneous bookkeeping would ruin the performance. Separate search of the linefeed (or carriage return) characters changes feels better from the user's point of view, because search is needed only, when an occurrence was found.

1.2 List of publications

The thesis consists of an overview and the following publications which are referred in the text by their Roman numerals [I] – [VII].

I Jorma Tarhio and Hannu Peltola: String matching in the DNA alphabet. *Software: Practice and Experience*, **27**(7):851–861, 1997. doi: 10.1002/(SICI)1097-024X(199707)27:7<851::AID-SPE108>3.3.CO;2-4

II Hannu Peltola and Jorma Tarhio: Alternative algorithms for bit-parallel string matching. In *Proceedings of the 10th International Symposium on Information Processing and Information Retrieval, SPIRE'03, Lecture Notes in Computer Science* **2857**:80–93, 2003. doi: 10.1007/978-3-540-39984-1_7

III Hannu Peltola and Jorma Tarhio: On string matching in chunked texts. In *Proceedings of the Conference on Implementation and Application of Automata, CIAA'07, Lecture Notes in Computer Science* **4783**:157–167, 2007. doi:10.1007/978-3-540-76336-9_16

[2]Grep is a command-line utility for searching plain-text data sets for lines matching a regular expression. The operation of Grep utility is defined in POSIX standard IEEE Std 1003.1.

IV Petri Kalsi, Hannu Peltola, and Jorma Tarhio: Comparison of exact string matching algorithms for biological sequences. In *Proceeding of the 2nd International Conference on Bioinformatics Research and Development, BIRD 2008, Communications in Computer and Information Science* **13**:417–426. Springer-Verlag, Berlin, 2008. doi: 10.1007/978-3-540-70600-7_31

V Branislav Ďurian, Jan Holub, Hannu Peltola, Jorma Tarhio: Tuning BNDM with q-Grams. In *Proceedings of ALENEX09, the Tenth Workshop on Algorithm Engineering and Experiments*: 29–37, 2009. ISBN: 978-0-898719-30-7 *URL:* http://www.siam.org/proceedings/alenex/2009/alx09_003_durianb.pdf

VI Branislav Ďurian, Jan Holub, Hannu Peltola, Jorma Tarhio: Improving practical exact string matching. *Information Processing Letters*, **110**(4):148–152, 2010. doi:10.1016/j.ipl.2009.11.010

VII Branislav Ďurian, Hannu Peltola, Leena Salmela, Jorma Tarhio: Bit-parallel search algorithms for long patterns. In *Proceedings of the 9th International Symposium on Experimental Algorithms, SEA 2010, Lecture Notes in Computer Science* **6049**: 129–140, 2010. doi: 10.1007/978-3-642-13193-6_12

1.3 Author's contribution

The author of this dissertation manuscript, M. Sc. Hannu Peltola, was involved in all the research activities leading to these publications, including innovation of new ideas, implementation, testing, and writing. The final implementations of the algorithms (except BMH2 and BMH2C in [IV] by Petri Kalsi) and test runs were made by the author including numerous implementations of other algorithms. The author has also made the necessary modifications to them. The present author and Jorma Tarhio have written together all the publications with an exception mentioned below.

Publication I: String matching in the DNA alphabet. This paper introduces an efficient q-gram method for the Boyer–Moore–Horspool algorithm.

Publication II: Alternative algorithms for bit-parallel string matching. This is our first study of bit-parallel string matching. The importance of simplification began to become clear. The author solely invented and developed all the variations of the SVM algorithms.

Publication III: On string matching in chunked texts. The article contains several improvements to the exact string matching on texts with autocorrelated and skew byte distributions.

Publication IV: Exact string matching algorithms for biological sequences. This paper is partly a continuation of Publication [I]. Experimental tests showed the relatively poor performance of several recent algorithms. Here we utilized the idea of reading several bytes simultaneously for the first time.

Publication V: Tuning BNDM with q-Grams. This article combines successfully four methods to families of algorithms: q-grams, bit-parallelism, simplification, and simultaneous processing of two bytes.

UFNDMq algorithm is based on the prototype codes SOFNDM4_ufast and SOFNDM4_uk developed by Branislav Ďurian. UFNDMq is not presented in the other papers.

Publication VI: Improving practical exact string matching. This is an updated and shortened journal version of the workshop article [V].

Publication VII: Bit-parallel search algorithms for long patterns. This article contains several novel algorithms for the searching for long patterns. Most of these algorithms are based on the ideas of Branislav Ďurian. Leena Salmela wrote the complexity analysis.

1.4 Organization

This thesis is organized as follows. Such exact string matching algorithms that are useful for understanding the novel ones, are presented in Chapter 2. There are also some notes about things that affect their performance. When designing new algorithms for the exact string matching, the so-called bit-parallelism has been popular approach during the last decade. Chapter 3 introduces algorithms, that use q-grams, but do not use bit-parallelism. In Chapter 4 several bit-parallel algorithms are presented. Solutions for searching in special kinds of texts are dealt with in Chapter 5. In Chapter 6, we discuss the testing of algorithms and present an experimental speed comparison of exact string matching algorithms.

2. Background

Our goal is to improve the practical speed of exact string matching. Currently many of the fastest algorithms utilize bit-parallelism. Another efficient idea is to start the comparison with the pattern by reading several text characters before testing them for the first time. Both approaches are more or less based on "classical" ideas and algorithms which are introduced in this section.

We start index from one, and therefore we denote the pattern $P = p_1 p_2 \dots p_m$ and the text $T = t_1 t_2 \dots t_n$.

2.1 Boyer–Moore algorithm

Typically, exact string matching is started by aligning the pattern with the beginning of the text. Then the pattern is compared with the corresponding text until a match or a character mismatch is found. Next, the pattern is moved forward in the text. Obviously it is advantageous to move more than one position at a time, whether some of these comparisons are unnecessary and can be thus avoided. In the best case, this enables "sublinear"[1] time complexity. The first algorithm that took advantage of this approach is the Boyer–Moore algorithm [BoM77]. Quite many practical exact string matching algorithms are based on it [ChLe04].

The most characteristic feature in the Boyer–Moore algorithm is the comparison order of character pairs between the pattern and the text: it starts from the end of the pattern and continues backwards. The intention of the comparison order is to enable in a straightforward manner a long shift of the pattern along the text to the next possible position. The orig-

[1] Strictly speaking time complexity $\mathcal{O}(n/m)$ is not sublinear: When the text length doubles, also the expected search time doubles. However, it is quite common to use loose speak in this exact string matching case. Perhaps this comes from the original article [BoM77], where sublinear is in quotation marks.

inal Boyer–Moore algorithm can be formulated as Algorithm 1. (Boyer and Moore gave a version that searches only the first occurrence. The presentation of **BM_orig** is according to Knuth [KMP77, p. 341].)

Algorithm 1 BM_orig($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
1:  Initialize δ1[*] and δ2[*]
    /* Searching */
2:  k ← m
3:  while k ≤ n do
4:      j ← m
5:      while j > 0 and t_k = p_j do /* and is conditional */
6:          j ← j − 1;   k ← k − 1
7:      if j = 0 then
8:          match found at k
9:          k ← k + m + 1
10:     else
11:         k ← k + max{δ1[t_k], δ2[j]}
```

The operation of the algorithm can be understood as the sliding of the pattern along the text. At every alignment it is checked whether the pattern matches the text. Shifting of the pattern is based on two heuristics that are implemented in the shift tables δ_1 and δ_2. The first heuristic produces the *occurrence shift*. It is implemented as the shift table δ_1. It is often also called the *bad-character shift*. The basic idea is to skip fast over parts of the text that cannot contain a match. If the text character at the end of the pattern does not occur at all in the pattern, it is possible to shift the pattern forward m positions. So, in the best case the algorithm has to check only n/m characters of the text. If the text character on the other hand appears in the pattern, one can shift the pattern by the minimal length concordant with the matching character. Formally, the occurrence shift can be defined for each character x of the alphabet by:

$$\delta_1[x] = \min\{s | s = m \text{ or } (0 \leq s < m \text{ and } p_{m-s} = x)\}$$

Let us suppose that on pairwise comparison of the pattern and the text, all the previous pairs have matched, but then a mismatch is encountered. Then the shift can be based on the idea that after the shift the whole previously matching suffix should be aligned compatibly to the new pattern position. This happens with the previous occurrence of the same pattern suffix, or if there is not any, then with the longest prefix of the pattern that

matches the matching suffix part. In addition, there must be a change in the pattern character that initially caused the mismatch. This heuristic is called the *matching shift* or sometimes also the *good-suffix shift*[2]. Formally the matching shift can be defined as follows [KMP77, p. 342]:

$$\delta_2[j] = \min\{s + m - j \mid s \geq 1 \quad \text{and} \quad (p_{j-s} \neq p_j \text{ or } s \geq j) \quad \text{and} \quad ((p_{k-s} = p_k \text{ or } s \geq k) \text{ for } j < k \leq m)\}$$

for $1 \leq j \leq m$. The matching shift can be extended also to the case when a match is found. Then $\delta_2[0]$ is equal to $\delta_2[1]$. The verifying order in alignments is not necessarily from right to left. Sunday [Sun90] introduced a version of the matching shift that works for any fixed permutation of the comparison positions in alignments.

After each mismatch, the original Boyer–Moore algorithm chooses the larger shift given by the two heuristics. The algorithm also forgets all the characters examined thus far.

2.2 Skip loop

During execution of the Boyer–Moore algorithm relatively many conditional branches are encountered. This phenomenon constitutes a challenge in pipeline processors. If something is known or assumed about the character distribution (e.g. the alphabet), it is possible to reformulate the algorithm to a more efficient form. Instead of trying every alignment of the pattern, the Boyer–Moore algorithm skips over typically most of them. This idea brings efficiency in practice. We use the term *skip loop* to indicate various looping methods for skipping past immediate mismatches in the text. The term comes from Hume and Sunday [HuS91], who brought attention to this too often neglected idea of Boyer and Moore [BoM77, p. 765].

Fast loop

When already the last character of the pattern and the corresponding text character do not match, $\delta_2[m]$ gives a shift that is at most as large as $\delta_1[]$. (A formal proof is given by Cantone and Faro [CaF03].) Thus, it is advantageous to use only the occurrence shift (instead of unnecessarily maximizing with matching shift) for the last character of the pattern.

[2] Richard Cole [Col91] was probably the first one to use the terms bad-character shift and good-suffix shift.

Already Boyer and Moore noticed that on natural language in general a randomly picked text character either does not exist in the pattern or at least is not the same as p_m. This assumption holds if both the last character of the pattern is not fairly common in text and the text does not have some special unfavorable internal structure.

Boyer and Moore carefully studied the implementation details of their exact string matching algorithm and gave an enhanced version of the algorithm [BoM77, p. 765] containing a skip loop structure that they called *fast loop*. Algorithm 2 **BM_fast** implements the same idea in a more block structured style. In the BM_orig algorithm in every alignment (except the last one) of the pattern at least two tests were made: has the end of the text already passed (line 3 in the Algorithm 1), and does the last text character in this alignment match with p_m (the second test on line 5)? In BM_fast, only the end of the text is tested. The basic idea is to set an

Algorithm 2 BM_fast($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
    /* Preprocessing */
 1: Initialize δ1[*] and δ2[*]
 2: for all c ∈ Σ do
 3:     δX[c] ← δ1[c]
    /* Searching */
 4: large ← n + m + 1;   δX[pm] ← large
 5: k ← m
 6: while k ≤ n do
 7:     repeat
 8:         k ← k + δX[tk]
 9:     until k > n                          /* end of fast loop */
10:     if k ≤ large then
11:         return
12:     k ← k − large − 1
13:     j ← m − 1
14:     while j > 0 and tk = pj do /* and is conditional */
15:         j ← j − 1;   k ← k − 1
16:     if j = 0 then
17:         match found at k
18:         k ← k + m + 1
19:     else
20:         k ← k + max{δ1[tk], δ2[j]}
```

artificially long shift for p_m in the occurrence shift. This modified shift is denoted here by $\delta_X[*]$[3]. If the length of the artificial shift is at least $m + n$, it is easy to distinguish the termination of the text from the artificially long shift caused by the match. To continue the comparison in the alignment, an equally long shift backward is made first. Boyer and Moore called this verifying phase *slow loop*. The fast loop is actually a while loop testing for the end of the text and containing shifting the pattern according to $\delta_X[*]$.

In the implementation of string matching algorithms, there is a risk of dangling pointers or indexing out of bounds of an array. The ANSI C standard requires, that pointers may be compared only if they point inside or immediately after the same array of characters. To work safely with the fast loop using pointers, one should reserve after the text extra (unused) space at least for the $n + m$ characters. (Space requirement depends on units, which are used in accessing of the text: in practice bytes or 16-bit words.)

Many implementations of the Boyer–Moore algorithm let pointers (that are pointing to the text and the pattern) slide during pairwise comparison to the left of the alignment, and then use either of those pointers. This may be a non-conforming use of C, especially when a match is located in the beginning of the text. This can be avoided by placing the text so that it starts from the second position of the text array. The same applies also to the pattern. Thus while accessing characters text characters using pointers in fast loop at least $2n + m + 1$ bytes should be allocated for the text array. (Actually, in virtual memory the actual cost caused by these space reservations is very small.)

Least cost loop

While testing the fast loop, Hume and Sunday [HuS91] realized that a large part of the performance depends on how long the algorithm stays in the skip loop. Since the pattern can be scanned in any order, the skip loop character could be any character of the pattern. With a rare skip character we will, on the average, fall out of the skip loop much more seldom than with more common characters. (See also related discussion later in Subsection SLFC.) Naturally, this approach requires at least partial knowledge about the character frequencies in the text. If the skip loop character is not the same than the last character of the pattern, then the

[3]Boyer and Moore called δ_X as δ_0. Here δ_0 is used for other purpose.

shifts are shorter on the average. In practice, the shifting is performed in a similar fashion as with accordingly truncated pattern.

When the probabilities of characters in the text are known, we can estimate the expected work done in the slow loop. On the other hand it is possible to estimate the expected skip distance. If the average costs of the skip loop and the slow loop are known on a given computer, it is possible to calculate the pattern character with the smallest expected cost. Hume and Sunday use in their implementation the expected skip distance in a random string instead of the real skip distances with given particular pattern. They named this kind of skip loop the *least cost loop*, or *lc* for short [HuS91, pp. 1228–1231].

The relative speed of conditional branches varies among computers. Therefore, the choice of a character position depends also on the hardware. So it is useful to utilize constant parameter such as t_{slow} (`tcmp` in their program codes), which is adjusted by running a calibration program for given computers.

In Publication [III], it was shown that there are cases where it is beneficial to use a character different from the last character of the pattern as test character.

Unrolled fast loop

It is also possible to reduce the number of tests required to know whether the end of the text has been reached. If a copy of the pattern is placed as a *stopper* after the text, any kind of skip loop will terminate. Instead of the whole pattern also m copies of the test character (i.e. p_m in other than the least cost loop) would work out as well, if just a single test character is used. As was the case with the fast loop, it is possible that after every shift the skip loop ends and the algorithm enters the slow loop. However, in practice, several shifts on the average are carried out before exiting the skip loop. In Subsection SLFC, we will discuss more precisely, how frequently the control will exit the skip loop.

Setting the shift for p_m in $\delta_1[\,]$ to zero enables both, unrolling the skip loop and exiting it, when a potential match or the stopper after the end of the text is encountered. Hume and Sunday [HuS91] named this kind of a skip loop the *unrolled fast loop* or *ufast loop* for short. The idea has been used before, but Hume and Sunday introduced it to the broader scientific community. Algorithm 3 **BM_ufast** implements a 3-fold unrolling factor.

The common unrolling factor 3 is based on Table 2.1 and the following

Algorithm 3 BM_ufast($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
    /* Preprocessing */
 1: Initialize δ1[*] and δ2[*]
 2: for all c ∈ Σ do
 3:     δ0[c] ← δ1[c]
    /* Searching */
 4: δ0[p_m] ← 0
 5: t_{n+1} ... t_{n+m} ← P
 6: k ← m
 7: while k ≤ n do
 8:     s ← δ1[t_k]
 9:     while s ≠ 0 do /* in C plain s suffices in condition */
10:         k ← k + s; s ← δ0[t_k]
11:         k ← k + s; s ← δ0[t_k]
12:         k ← k + s; s ← δ0[t_k]        /* this is ufast3 loop */
13:     if k > n then
14:         return
15:     k ← k − 1
16:     j ← m − 1
17:     while j > 0 and t_k = p_j do /* and is short circuit */
18:         j ← j − 1;   k ← k − 1
19:     if j = 0 then
20:         match found at k
21:         k ← k + m + 1
22:     else
23:         k ← k + max{δ1[t_k], δ2[j]}
```

reasoning [HuS91, p. 1228]: "The benefits of unrolling are substantially dependent on the length of the patterns and the system design, for example, the size of the instruction cache. After measuring different unrolling factors (with match=*fwd*, shift=*inc*), we picked 3-fold unrolling as the best compromise across systems." *'fwd'* means that pairwise comparison was performed from the left to the right. The pattern was moved always one position forward, and therefore pairwise comparison was tried at all alignments in the text.

The pattern set and the text are `rand.500` and `bible` by Hume and Sunday [HuS91]. This becomes evident from the reported average shift lengths and the numbers of examined characters. It should be noted that

shift=*inc* means shifting just one position after verification [HuS91, 4.3.1]; in the skip loop the δ_1 shift is used. The *lc* shifts equally to *fast*, when the test character is the last character of the pattern. In *lc*, the verification is a little more complicated than in *fast*, and in order to simplify it, the test character will be checked again. Actually, in most cases, the test character in *lc* is the last character of the pattern as in *fast*: with the 386 processor 67 patterns out of 500 and with Sparc only 35 out of 500 test characters were different. Clearly, the performance of *lc* was better with the higher values of parameter t_{slow}. It is possible that still higher values of t_{slow} would have produced still better speed.

On the average, patterns were shifted 191629.1 times and 10178.6 times drifted to the slow loop while using *fast* shift. With the *lc* skip, patterns were shifted on 386 processors on the average 203998.1 times, but only 8971.0 times ended to the slow loop; on Sparc, there were 202458.8 shifts and 9572.1 visits in the slow loop.

Obviously, while using the ufast loop the average number of zero length shifts increases, when the unrolling factor increases. Also, the statistics reprinted in Table 2.1 show this clearly. It is evident that the performance of ufast style unrolling depends on the character distribution. In this case the relative frequency of the last character of the pattern is a fundamental element. On the average, the higher frequency, the smaller the optimal unrolling factor is.

Table 2.1. Execution speeds with fast and lc loops and various unrolling factors listed in Hume and Sunday [HuS91, p. 1228 and 1231]. The column cmp+jump shows the total number of text characters read in the skip loop (jump) and pairwise comparison (cmp).

Algorithm	Execution Speed (MB/s)						Statistics	
	386	sparc	mips	vax	68k	cray	step	cmp+jump
fast	2.42	6.73	10.92	5.13	3.41	7.68	5.22	202619 (20.3%)
lc	2.27	7.13	12.57	5.71	4.28	9.23	4.93	212909 (21.3%)
unroll1	1.81	6.83	11.91	5.30	4.02	8.37	5.22	202620 (20.3%)
unroll2	2.42	7.06	12.45	5.63	4.12	9.05	5.08	207753 (20.8%)
unroll3	2.66	7.12	12.54	5.84	4.20	9.21	4.95	213048 (21.3%)
unroll4	2.79	7.08	12.48	5.84	4.20	9.16	4.82	218468 (21.8%)
unroll5	2.84	7.00	12.39	5.87	4.15	9.04	4.69	224007 (22.4%)
unroll6	2.86	6.92	12.25	5.79	4.11	8.90	4.57	229648 (23.0%)
t_{slow}	4.91	3.04	3.29	3.86	2.97	3.34		

The best value of the unrolling factor depends on the pattern, the text, and the computer architecture. The effect of the compiler is perhaps quite small, because this skip loop is so simple. If the control often leaves the skip loop, a smaller factor is better. Also the performance deviation in 32-bit and 64-bit modes on modern Pentium processors in Publication [V] encouraged us to examine the performance of various skip loops by repeating tests in Table 2.1. The results are summarized in Table 2.2. Specifications of the test computers A, B, C, D, and H used in the test are given in Appendix B.

Table 2.2. Execution speeds with various unrolling factors with newer machines.

Algorithm	Execution Speed (GB/s)								
	A	B		C		D		H	
	32	32	64	32	64	32	64	32	64
fast	.206	1.21	1.21	1.25	1.28	1.56	1.60	.307	.318
lc	.201	1.14	1.02	1.23	1.13	1.55	1.50	.334	.316
unroll1	.206	1.17	1.02	1.26	1.16	1.61	1.49	.285	.262
unroll2	.200	1.17	1.04	1.24	1.15	1.56	1.50	.319	.306
unroll3	.199	1.13	1.01	1.22	1.13	1.54	1.51	.332	.316
unroll4	.198	1.09	.976	1.18	1.10	1.51	1.49	.337	.320
unroll5	.196	1.07	.960	1.18	1.08	1.48	1.46	.338	.323
unroll6	.194	1.05	.944	1.16	1.07	1.47	1.44	.337	.319
unroll7	.193	1.03	.920	1.13	1.05	1.44	1.43	.335	.318
unroll8	.191	1.01	.905	1.12	1.04	1.41	1.40	.337	.316
unroll9	.189	.987	.882	1.10	1.02	1.39	1.38	.334	.315

Only computer H uses the Sparc architecture; all others run on the X86 or X86_64 architectures. Table 2.2 shows that the performance of *ufast* depends clearly on the computer architecture. The *fast* and *lc* skip loops seem to be competitive.

One can expect that the performance of the *lc* skip loop is even better on customary texts than here, because the text and patterns in the test above contained no upper case letters.

It is also easy to make a *ufast* skip loop, where more than one test character from the pattern is used. Then, there is a smaller probability that control gets to the slow loop due to a single character match.

SFC

Quite shortly after the Boyer–Moore algorithm was published, Horspool examined "circumstances under which it should be employed" [Hor80]. He also studied alternative implementations using special purpose instructions.

In the 1980's many mainframe computers had instructions that were meant for character strings. A typical instruction did memory search for the first occurrence of the designated character: The IBM 360/370 series instruction set had the Translate and Test `TRT` instruction. It could search up to 256 bytes for a given character. The Burroughs B-series had the Search While Not Equal `SNEU`. The Univac 1100 series had the Search equal `SE` instruction [Hor80]. The VAX architecture had Locate Character `LOCC`[4] instruction. It could search up to 65535 bytes for a given character.

Horspool noted that, if an instruction for searching the next occurrence of the given character is available, it would be really handy to apply it as a skip loop. He named this approach *SFC loop* (Scan for First Character). If the available instruction cannot examine the whole text once, it is obvious to do the scan in parts with a tight loop around that instruction.

The x86-architecture provides versatile string instructions. The Scan String instruction (SCAS) is for searching of a byte (or also word, doubleword or quadword). The Compare String operand (CMPS) can be used for comparing the contents of two memory addresses. Those instructions can be preceded by the REP prefix for operations on longer blocks. (Quite often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made.) For example REPNE SCANB seeks for next occurrence of given byte (for at most given number of bytes), but REPE SCANB seeks for next byte that is unequal. These string instructions can be performed either backwards or forwards according to the setting of the direction flag (DF) flag.

The Intel Core microarchitecture and AMD K10 (K8L) contain SSE4 (Streaming SIMD Extensions 4) CPU instruction set. Subset SSE4.2 offers PCMPESTRI – Packed Compare Explicit Length Strings, Return Index – instruction than can be used for implementing *SFC loop*.

Of course it is possible to emulate the SFC loop by examining text

[4]On most newer VAXes `LOCC` was emulated and therefore relatively slow. Particularly VAX 8550, which was used by Hume and Sunday, had a non-VLSI hardware, and thus had `LOCC` implemented.

characters one by one. Another way to implement the idea is to use a suitable library function, e.g. `memchr` in C.

SLFC

Horspool observed also that SFC loop does not always "use the search instruction in the best possible way" [Hor80]. In natural languages, some characters are more common than others. On the average, the lowest frequency character in a pattern occurs relatively rarely in the text. A skip loop utilizing this idea is called *SLFC loop* (Scan for Lowest Frequency Character). Horspool has summarized the expected number of characters that are skipped before finding the lowest frequency character in the pattern [Hor80, Table I, p. 503]. Those numbers may seem amazingly large. The expected number of characters that are skipped can be approximated by the geometric distribution (because texts are of finite lengths). If the frequency of the given character is p, then the expected number of text characters not matching is $1/p$. Let us denote the probability of character c_i by p_i. Then, for randomly chosen character, the expected number of characters that are skipped is

$$E(Y) = \sum_i p_i \frac{1}{p_i}$$

This is actually the number of different characters appearing in the text! A collection of American English texts known as the Brown corpus has been widely used in studying language statistics. The alphabet of the corpus contains 94 characters [BCW90, p. 79].

Hume and Sunday tested SFC and SLFC with direct implementation and using `memchr` library routine, but they did not include in their testing any computer specific special instructions [HuS91, p. 1231]: "Provided the preprocessing to find the least frequent character is not onerous, SLFC is better than SFC. The benefits of using the `memchr` library routine are quite system specific." In the skip loop tests SFC and SLFC were always slower than lc, fast, and ufast, as expected. The tests of Hume and Sunday were performed with a set of various length patterns. Horspool reported the tests with pattern sets that had fixed pattern lengths. The programs were coded with 370/Assembler. There the SLFC skip loop was competitive on patterns up to five characters long [Hor80].

If the character distribution is far from uniform, then the probability of the least frequent character is lower and SLFC falls less frequently to the slow loop, when the pattern is longer.

The byte distribution may become skewed also in insidious ways. When

ASCII characters are represented in UTF-16 (16-bit Unicode Transformation Format) coding, on little-endian machines the rightmost bytes are zeros [III]. When there are plenty of characters that are not ASCII characters in a UTF-8 coded text, the byte distribution can be surprisingly different than e.g. from a Latin-1 (ISO 8859-1) coded text. This affects both shifting and the frequency of exiting the skip loop.

2.3 Boyer–Moore–Horspool algorithm

In 1980, Horspool published a versatile article about exact string matching [Hor80]. His main focus was on natural language texts. The previous subsection is largely based on his analysis on the effect of character frequencies. Horspool also analyzed the practical behavior of Boyer–Moore algorithm, and gave a simplified version of it. His central observation was that "the only purpose of δ_2 is to optimize the handling of repetitive patterns (such as 'XABCYYABC') and so to avoid a worst case running time of $\mathcal{O}(mn)$". So he designed Algorithm 4 **BMH**, which is a simplified version of the Boyer–Moore algorithm. It is also known as the Boyer–Moore–Horspool algorithm or the Horspool algorithm (or Hor in Publication [III]). (The original version searched only the first occurrence similarly to the original Boyer–Moore algorithm.)

A characteristic feature of the Boyer–Moore–Horspool algorithm is to

Algorithm 4 BMH($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
   /* Preprocessing */
 1: for all c ∈ Σ do
 2:     δ1[c] ← m
 3: for i ← 1 to m − 1 do
 4:     δ1[p_i] ← m − i
 5: lastch ← p_m
   /* Searching */
 6: k ← m
 7: while k ≤ n do
 8:     ch ← t_k
 9:     if ch = lastch then
10:         if t_{k−m+1} ... t_k = P then
11:             match found at k − m + 1
12:     k ← k + δ1[ch]
```

start in each alignment with the comparison of the last character of the pattern with the corresponding text character. In Algorithm 4 it is on line 9. We call it *guard test*, following Hume and Sunday [HuS91]. There can be several guard tests, and guard characters can be from any position of the alignment. Horspool used a special instruction for comparing the pattern and the text on line 10 [HuS91, p. 502, 505].

Over the years in many articles it has been claimed that the Boyer–Moore–Horspool algorithm performs more or different tasks than the original one. The following are some examples of the points that were attributed to the BMH algorithm:

- Comparison of the alignment is made backwards without the guard test. The remaining number of alignments is explicitly counted; so two values need to be updated instead of one.[5]

- The index points to text positions at the beginning of the alignment instead of the end of the alignment. Adding $m - 1$ makes it slower, when the last character pair does not match. Comparison of the rest of the alignment is made using the `memchr`. In tests of Hume and Sunday [HuS91, p. 1234], it was slower than self programmed loop.[6]

- Comparisons at the alignment are made backwards without the guard test. The last text character in the alignment is fetched again to evaluate the shift. [NaR02, Fig. 2.12]

- Comparisons at the alignment are made backwards without the guard test. Two tests in the comparison are replaced by only one by adding a character not appearing in the text before the pattern, and adding a character not appearing in the pattern before the text. Naturally this requires additional knowledge of the data. The last text character in the alignment is fetched again to evaluate the shift. [Bae89]

If it is possible to place a copy of the pattern (or m copies of the last character of the pattern) as a stopper beyond the end of the text, the `WHILE` loop in the Boyer–Moore–Horspool algorithm could be implemented with unrolled fast loop. The program codes by Hume and Sunday using the

[5] `http://en.wikipedia.org/wiki/Boyer-Moore-Horspool_algorithm`
[6] `http://www-igm.univ-mlv.fr/~lecroq/string/node18.html`

`md2` shift are implementations with this idea in mind [HuS91]. We have followed this approach by Hume and Sunday. Another alternative would be placing the stopper on the end of the text. Naturally the original end of the text should be saved first. After restoring the end of the text, (at most) $2m - 1$ positions from the end of the text have to be checked with some other method.

Cantone and Faro [CaF03] present a cross of the Boyer–Moore and the BMH algorithms. The outer skip loop utilizes only the BMH shift, and the shift by the matching heuristics is taken after each checking phase.

2.4 Sunday's Quick Search algorithm

Daniel Sunday has proposed an algorithm related to the Boyer–Moore–Horspool algorithm. He called it Quick Search algorithm [Sun90], which is shown as Algorithm 5 **QS**.

In **QS**, the shift is based on the text character immediately following the current alignment instead the last text character of the alignment. This is possible, because the pattern is moved in any case at least one position. Thus, the shift for Sunday's Quick Search algorithm is one position longer than for the Boyer–Moore–Horspool algorithm on other characters except possibly for the last character of the pattern: if the last two characters of the pattern are the same, then the shift for that character is the same ($= 1$). Otherwise, the shift for the last character of the pattern in Sunday's Quick Search algorithm is 1, and thus shorter. Obviously, the relative increase to the average shift length is larger on shorter patterns.

The fact that QS reads in every alignment one character more than e.g. Boyer–Moore and Boyer–Moore–Horspool algorithms is often overlooked.

The QS algorithm is quite popular on exact string matching comparisons. One reason for that might be that the original article [Sun90] contained the implementation in the C language.

Let us consider an alphabet where characters are statistically independent of each other, and where the characters in decreasing frequency order are a_1, a_2, a_3, ... Then the length for occurrence shift suffers from most common characters appearing in the end of the pattern. For the Sunday's QS algorithm the worst expected shift length is for pattern $\mathrm{a}_m \ldots \mathrm{a}_3\mathrm{a}_2\mathrm{a}_1$. For the Boyer–Moore–Horspool algorithm the worst expected shift length is for pattern $\mathrm{a}_{m-1} \ldots \mathrm{a}_2\mathrm{a}_1\mathrm{a}_1$. Actually the character appearing in the end of the pattern is irrelevant for the shift, but the most frequent character

Algorithm 5 QS($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do
 2:     TD1[c] ← m + 1
 3: for i ← 1 to m do
 4:     TD1[p_i] ← m − i + 1
    /* Searching */
 5: k ← 0
 6: while k + m ≤ n do
 7:     i ← 1
 8:     while i ≤ m and t_{k+i} = p_i do /* and is short circuit */
 9:         i ← i + 1
10:     if i = m + 1 then
11:         report occurrence at k + 1
12:     k ← k + TD1[t_{k+m+1}]
```

maximizes the probability of falling to the slow loop.

In this version the verification of the match is made forward; basically it could be made in any order. However, every time the shift is one position, the last position in the alignments matches always. Therefore, if p_m is a really common character, the verification backwards (without a guard) is doubtful.

Longer shifts

In the Boyer–Moore algorithms the maximum function does not take out all the information behind $\delta_1[\,]$ and $\delta_2[\,]$ heuristics.

Apostolico and Giancarlo [ApG86] suggested a 2-dimensional shift table, that was indexed by the mismatch position index, and the text character that caused the mismatch. It produces at least as long shifts than in the original Boyer–Moore algorithm. Their other contribution was that the number of character comparisons was bounded by $2n$ on algorithm utilizing that 2-dimensional shift table.

If some frequent characters appear in the pattern, the expected shift length from the $\delta_1[\,]$ shift is smallish. Use of the shift table based on two adjacent text characters was suggested by Zhu and Takaoka [ZhT87]. The $\delta_1[\,]$ shift of the Boyer–Moore algorithm was replaced by a 2-dimensional shift table, that was indexed by the text character that caused the mismatch and the text character before it.

Cantone and Faro [CaF05] have presented the Forward-Fast-Search algorithm, which used a new kind of shift table. It was also 2-dimensional, but was indexed by the mismatch position index and the text character beyond the current alignment. So the text character position is related to Sunday's QS algorithm. The shifts are relatively clearly longer with short patterns. The use of lookahead character gives advantage on small alphabets.

2.5 Introduction to bit-parallel algorithms

Parallelism is a powerful way to boost computation. With just one processor and without vector instructions, it is sometimes possible to use *Single Instruction, Multiple Data (SIMD)* technique. It is achieved with the bit-parallelism where several values are updated simultaneously in a register. (The term bit refers to bitwise operations.) Boolean variables represent the most simple variable type. Probably the earliest use of bit-parallel techniques can be traced back to Dömölki [Döm64] already in 1964. Another article describing bit-parallel techniques for integers surfaced in 1975 by Lamport [Lam75]. Allison and Dix [AlD86] presented a bit-parallel algorithm for the longest common subsequence problem. As far as we know, it was the first bit-parallel string processing algorithm. A few years later, Baeza-Yates and Gonnet presented several variations of bit-parallel string algorithms [BaG92]. Their Shift-Or algorithm for exact string matching is fast for short patterns [NaR02, Fig. 2.22, p. 39].

Bit-parallel algorithms can also be interpreted as the simulation of an automaton – typically a nondeterministic one. The update operations for all states should be identical; otherwise we must first perform certain operations on a set of states, and then certain operations on some other set of states. Operands are typically called *bit-vectors*, and the essential bit-vector containing the state of the automaton is called the *state vector*. For good practical performance, the states of the whole automaton should fit into a register of a computer. Larger registers in modern processors allow simultaneous processing of several variables (states) packed into a single computer word. A natural data type for this purpose is the unsigned integer. It is often silently assumed that signed integers are represented in the two's-complement form; especially that the constant -1 has all bits set.

Another important bit-parallel algorithm is BNDM (Backward Non-

deterministic DAWG[7] Matching) [NaR98]. It simulates a nondeterministic automaton, and uses shifting related to the Boyer–Moore approach. BNDM is discussed in detail in Section 2.7.

A useful feature of bit-parallel string matching algorithms is the easiness and effectiveness in use of character classes in matching [BaG92]:

- Classes of characters: a given character set, complement, and don't care symbol.

- Ignore case. This scheme surely works only when one byte represents a character and the alphabet is known. Handling of ligatures and digraphs[8] mostly work correctly. A well known exception is the German esszett ligature (also called the scharfes s (sharp s)) ß, which expands when uppercased to the sequence of two characters "SS".

  These straightforward techniques do not expand to the Unicode character encoding, when the matching of the pattern and the text is done byte by byte: e.g. mathematical, musical, and currency symbols do not have uppercase, titlecase[9], and lowercase versions.

We use the following notations. The register width (or word size informally speaking) of a processor is denoted by w. If not otherwise stated we assume that the rightmost bit of the computer word represents the value $2^0 = 1$. A bit mask of s bits is represented as $b_s \cdots b_1$. The most significant bit is on the left. To distinguish the normal unsigned integers from binary numbers base 2 is used in marked as subscript in the end of binary number. A superscript stands for bit repetition (e.g. $10^2 1_2 = 1001_2$) i.e. run-length encoding. The C-like notations are used for bit operations: "|" bitwise (inclusive) OR, "&" bitwise AND, "^" bitwise exclusive OR, "~" one's complement, "<<" bitwise shift to the left with zero padding, and ">>" bitwise shift to the right with zero padding. For the shift operations, the first operand is unsigned and the second operand must be nonnegative and less than w.

[7] DAWG – Directed Acyclic Word Graph
[8] http://unicode.org/faq/ligature_digraph.html
[9] http://unicode.org/reports/tr21/tr21-3.html

2.6 Shift-Or

The Shift-Or algorithm is a short and simple bit-parallel string matching algorithm. The name comes from the operations used in updating the state vector: first the state vector is shifted, and then an Or operation is applied between it and and the bit-vector of the corresponding character in the current text position. Bit-vectors in table C have clear bits on positions, where the corresponding character occurs in the pattern. Shift-Or reserves only one bit per pattern character in the state vector D.

The **Shift-Or** algorithm for exact string matching is shown as Algorithm 6. Sometimes it is also called Bitap. This name comes from the generalization that is used in the source code of the agrep tool for approximate string matching [WuM92]. The time complexity of the Shift-Or algorithm is $\Theta(n \cdot \lceil m/w \rceil)$. This algorithm is on its best with quite small values of m. Then the time complexity of the Shift-Or algorithm is $\Theta(n)$, when $m \leq w$. Therefore, the search speed does not depend of pattern length m, but in practice the number of found occurrences affects slightly.

Algorithm 6 Shift-Or($P = p_1 p_2 \cdots p_m, T = t_1 t_2 \cdots t_n$)

Require: $m \leq w$

```
    /* Preprocessing */
 1: for all c ∈ Σ do C[c] ← (~0) >> (w − m)                    /* 1_2^m */
 2: for j ← 1 to m do
 3:     C[p_j] ← C[p_j] & ~(1 << (j − 1))                      /* 1^{m−j}01_2^{j−1} */
    /* Searching */
 4: epos ← 1; D ← ~0; mm ← 1 << (m − 1)                        /* 10_2^{m−1} */
 5: while epos ≤ n do
 6:     D ← (D << 1) | C[t_epos]
 7:     if (D&mm) ≠ mm then /* = 0 does the same */
 8:         report an occurrence at epos + 1 − m
 9:     epos ← epos + 1
```

The original version of the Shift-Or algorithm [BaG92, Fig. 4, p.78] uses a loop for scanning for the first character of the pattern (SFC loop), but it is often omitted, e.g. [NaR02, Fig. 2.6, p. 20], [Smy03, Fig. 7.4.1, p. 204], [FrG05, Alg. 1, p. 378].

The placement of the actually used successive m bits in bit-vectors not fixed; left (least significant bits) or right are most useful. Because we assume that shift operations have zero padding, it is possible to for-

mulate the algorithm so that the shifting (in the search phase) is performed to the right instead of left. Naturally, the bit-vectors based on the pattern must be reversed. Similarly, the Or operation can be replaced with And; then the algorithm actually becomes **Shift-And**. Care should be taken that the filling bits produced in shift operations are suitable. Correcting them for example in the following way demands extra work:

6: $D \leftarrow (D << 1 \mid 1) \ \& \ C[t_{\boldsymbol{epos}}]$

Checking for a match (on line 7) is an important part of the innermost loop. A small change in testing may cause a large change to practical performance. Fredriksson and Grabowski [FrG05] noticed that the unused bits of the state vector could be utilized: found matches can be stored and then checked for several positions at same time. Instead of a single one character at a time being processed, U consecutive text characters are processed, and then the non-existence of matches can be checked from U consecutive alignments. This is beneficial, if there are clearly less occurrences of the patterns than at every Uth position on the average. This idea has been implemented in Algorithm 7 called **Fast-Shift-Or** [FrG05, Alg. 4, p. 381], [FrG09, Alg. 4, p. 583][10].

The shifts on lines 1 and 2 are made in two parts, because the shift length must be less than w. If $U > 1$, shifts can made in one operation:

1: **for all** $c \in \Sigma$ **do** $C[c] \leftarrow (((\sim 0) >> (w - U + 1))$ /* 1_2^{U-1} */

2: $<< m) \wedge (\sim 0)$ /* $0^{U-1}1_2^m$ */

The Fast-Shift-Or algorithm may examine at most $U - 1$ characters after the last text character. This can be easily avoided by exiting while loop earlier and processing the rest with the Shift-Or algorithm. Alternatively the possible found erroneous matches can be rejected during check phase.

Fredriksson and Grabowski report that Fast-Shift-Or gives about a $2 - 5$ times speedup compared to the standard Shift-Or [FrG05]. Their original implementation just counted the number of matches, but did not determine their exact locations. Fredriksson and Grabowski noticed that the For-loop starting on line 7 of Algorithm 7 is automatically inlined by compilers for a small constant U.

The Fast-Shift-Or algorithm checks the state vector so rarely that other

[10] In original versions the meaningful bits were located to the left end of the bit-vectors, but here they are on the right. The correct initialization of the state vector D in the original versions is $(\sim 0) << (w - U - m)$.

Algorithm 7 Fast-Shift-Or($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $U \geq 1$ and $m + U - 1 \leq w$

```
    /* Preprocessing */
 1: for all c ∈ Σ do C[c] ← (((((~0) >> (w − U)) >> 1)          /* 1_2^{U−1} */
 2:       << (m − 1)) << 1) ∧ (~0)                              /* 0^{U−1}1_2^m */
 3: for j ← 1 to m do
 4:       C[p_j] ← C[p_j] & ~(1 << (j − 1))                     /* 1^{m−j}01_2^{j−1} */
    /* Searching */
 5: epos ← 1; D ← ~0; mm ← ((~0) >> (w − U)) << (m − 1)        /* 1^U 0_2^{m−1} */
 6: while epos ≤ n do
 7:       for j ← 0 to U − 1 do
 8:             D ← (D << 1) | C[t_{epos+j}]
 9:       if (D&mm) ≠ mm then
10:             Check and report occurrences
                at positions epos + 1 − m..epos + U − m
11:       epos ← epos + U
```

related bit-parallel string matching algorithms have to examine considerably less text characters to achieve even the same performance.

2.7 BNDM

An elegant way of reaching the asymptotic optimum average time complexity is the Backward DAWG Matching algorithm (BDM) [CrR94]. Nevertheless, the algorithm is complicated to implement and it is not fast enough for practical text searching tasks [NaR00, pp. 29–30]. Its asymptotic optimality is reached only when searching for very long patterns. The Backward Oracle Matching algorithm [ACR99, FaL08], a simplified version of BDM, is faster in practice. Another faster variation is BNDM (Backward Nondeterministic DAWG Matching) by Navarro and Raffinot [NaR00]. BNDM is a cross between the BDM and Shift-Or [BaG92] algorithms. The idea is similar to that of BDM, but instead of building a deterministic automaton, a nondeterministic automaton is simulated with bit-parallelism even without constructing it.

In BNDM [NaR00] (see Algorithm 8) the precomputed table $B[\,]$ associates each character with a bit mask expressing its locations in the pattern. (Table $B[\,]$ and table $C[\,]$ in the Shift-Or algorithm are logical complements on essential parts.) At each alignment of the pattern, the

Algorithm 8 BNDM($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $1 < m \leq w$

/* Preprocessing */

1: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow 0$
2: **for** $j \leftarrow 1$ **to** m **do**
3: $\quad B[p_j] \leftarrow B[p_j] \mid (1 << (m - j))$

/* Searching */

4: $i \leftarrow 0$
5: **while** $i \leq n - m$ **do**
6: $\quad j \leftarrow m; \quad last \leftarrow m; \quad D \leftarrow (\sim 0) >> (w - m + 1)$ /* 1_2^{m-1} */
7: $\quad$ **while** $D \neq 0$ **do**
8: $\quad\quad D \leftarrow D \;\&\; B[t_{i+j}]; \quad j \leftarrow j - 1$
9: $\quad\quad$ **if** $D\&(1 << (m-1)) \neq 0$ **then**
10: $\quad\quad\quad$ **if** $j > 0$ **then**
11: $\quad\quad\quad\quad last \leftarrow j$
12: $\quad\quad\quad$ **else**
13: $\quad\quad\quad\quad$ report occurrence at $i + 1$
14: $\quad\quad D \leftarrow D << 1$
15: $\quad i \leftarrow i + last$

algorithm reads the text from right to left until the whole pattern is recognized or the processed text string is not a substring of the pattern. Between alignments, the algorithm shifts the pattern forward to the start position of the longest found prefix of the pattern, or if no prefix is found, over the current alignment. With the bit-parallel Shift-And technique the algorithm maintains a state vector D, which has one in each position where a substring of the pattern starts such that the substring is a suffix of the processed text string. The standard BNDM works only for patterns which are not longer than w.

The inner while loop of BNDM checks one alignment of the pattern in the right-to-left order. At the same time the loop recognizes prefixes of the pattern. The leftmost one of the found prefixes determines the next alignment of the algorithm.

The Boyer–Moore algorithm stops pairwise comparison of the pattern and the text after the whole pattern matches or the examined characters are not a suffix of the pattern. The BNDM algorithm does not stop that early: it continues as long as the examined characters form a substring of the pattern. Therefore at a given alignment the BNDM algorithm may

examine more text characters than the Boyer–Moore algorithm. On the other hand, the BNDM algorithm takes at least as long shifts as the Boyer–Moore algorithm, because BNDM finds always the longest suffix of the alignment, which is a prefix of the pattern.

2.8 Recombinations and reinventions of ideas

Several research papers are published every year about exact string matching. Often it is quite hard to judge the value of their findings in practical situations. If an algorithm works better in some situation, it may also work slower in several other more common cases. Many useful innovations have been found out on some other—often related—research field. The (large) number of publications makes it quite hard to be informed of all of them. Some ideas are also not clearly expressed. Therefore many ideas have been invented several times. Next some examples are presented.

If the alphabet is small or if some characters are very common, it may be advantageous to process more than one character together. Already Knuth [KMP77, p. 331] mentioned this idea.

Starting of a slow loop is typically quite laborious (compared to a skip loop). Therefore it might be faster to add some additional test before starting complete pairwise comparison of the pattern and the text. This idea appears for example in [Rai92, Rai99, SSA04, TVS06].

The Knuth–Morris–Pratt algorithm [KMP77] has linear performance also with potential occurrences while for example the Boyer–Moore algorithm [BoM77] works faster while running in a skip loop. It is tempting to combine these features. The Knuth–Morris–Pratt algorithm was merged with the Boyer–Moore–Horspool algorithm by Baeza-Yates [Bae89w], and with the Boyer–Moore algorithm in Exercise 8.2.4. by Smyth [Smy03, pp. 211–212], and also with with Sunday's Quick Search algorithm [FJS07].

On the other hand, some useful suggestions are frequently omitted. For example the BM_fast is rarely included in speed tests although its is faster in practice than the BM_orig.

3. Use of q-gram fingerprints

3.1 Introduction

While applying a Boyer–Moore type algorithm [BoM77], most often only the occurrence heuristic is applied for shifting. Algorithms of this type are greedy in the sense that the pattern is moved forward after the first character mismatch of an alignment is observed. Shifts by occurrence shift may then be unnecessarily short, if shifting is based on a single character. On the other hand, the probability of having the algorithm to enter to the slow loop is almost always higher, when the decision is based on a single character. (See Section 2.2.) Therefore, it is advantageous to apply strings of the q characters, instead of single characters. This technique was already mentioned in the original paper of Boyer and Moore [BoM77, p. 772], and Knuth [KMP77, p. 341] analyzed theoretically its gain. Zhu and Takaoka [ZhT87] presented the first algorithm utilizing the idea. Their algorithm uses two characters for indexing a two dimensional array. They also gave another version based on hashing. This section describes methods that are especially useful with small alphabets such as DNA.

A *q-gram*, (or a q-tuple) is a continuous substring of q characters. In a way, a q-gram represents a character of a larger alphabet.

Baeza-Yates [Bae89] introduced an extension to the Boyer–Moore–Horspool algorithm [Hor80] where the shift array is indexed with an integer formed from a q-gram with `shift` and `add` instructions. For this kind of approach, the practical upper limit with 8-bit characters is two characters, because $q \cdot 8$ bits are needed. Thus, the shift table would contain 2^{24} items for $q = 3$.

Berry and Ravindran [BeR99] suggested a 2-gram algorithm that was inspired by the Sunday's Quick Search algorithm [Sun90] (Algorithm 5).

The shift is based on a two-dimensional table, which is indexed with the two text characters following the current alignment. The Berry–Ravindran algorithm suffers from a frequent character at the end of the pattern, because in that case, the shift is only one position. One may think that this happens so rarely that the slowdown is insignificant. In Publication [IV], it was shown that both the expected shift length and the practical performance suffer from this greedy approach. It was also shown how to fix the inefficiency of the Berry–Ravindran algorithm by using more conservative shifting. In the modified Berry–Ravindran algorithm the shifts are based on the last character of the current text alignment t_s and the next character t_{s+1}, instead of t_{s+1} and t_{s+2} as in the original Berry–Ravindran. The maximum shift in the original algorithm is $m + 2$, and in the modified version $m + 1$. Let us assume that text characters are statistically independent. With the discrete uniform distribution of c different characters, the probability of a shift of the pattern by one position is approximately $1/c$ for the original and $1/c^2$ for the modified algorithm. This is due to the inherent weakness of Sunday's QS algorithm [Sun90]: if the last character of the pattern is common, then the probability of a shift of one is high ($= 1 - 1/c$). In such a case, t_{s+2} is useless in the computation of shift in the Berry–Ravindran algorithm. The expected shift length of the modified version is longer. This is clearly visible in practical tests especially with small alphabets. Our test results in Publication [IV] show that this weakness of is noticeable even with amino acids.

For the DNA alphabet, Kim and Shawe-Taylor [KiS94] introduced an alphabet compression by masking the three lowest bits of ASCII characters. In addition to the `a`, `c`, `g`, and `t`, one gets distinguishable codes also for `n` and `u`. Even the important control code such as `\n`=LF has distinct value, but `\r`=CR gets the same code as `u`.[1] With this method they were able to use q-grams of up to six characters. Indexing of the shift array was similar to Baeza-Yates' approach [Bae89].

With a small alphabet the probability of an arbitrary short q-gram appearing in a long pattern is high. This restricts the average shift length. Kim and Shawe-Taylor [KiS94] introduced a boosting variation for the cases where the q-gram in the text occurs in the pattern. Two additional characters are checked one by one to achieve a longer shift.

[1] There are codes also for other incompletely specified bases in nucleic acid sequences except `n`, but they are very rare: *URL:* http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html

In most cases, the q-gram that is taken from the text does not match with the suffix of the pattern, and the pattern can be shifted forward. For efficiency one should use a skip loop [HuS91]. The easiest way to implement this idea is the unrolled fast loop.

3.2 Fingerprint method

As stated above, the Boyer–Moore algorithm is able on the average to take longer shifts by examining a q-gram at a time instead of a single text character [BoM77, KiS94, Bae89, Sed83, ZhT87]. The Baeza-Yates algorithm [Bae89, KiS94] uses this idea. We will consider variations of the Baeza-Yates algorithm and suggest implementation techniques, which will improve the total speed of the algorithm in the case of the DNA alphabet.

From a q-gram $a_0 \dots a_{q-1}$ we compute a *fingerprint* $id = \sum_{i=0}^{q-1} c^i \cdot a_i$, which is a (reversed) number of base c, where c is the size of the alphabet.

At each alignment of the pattern, the last q-gram of the pattern is compared with the corresponding q-gram in the text by testing the equality of their fingerprints. If the fingerprints match, we have found a *potential occurrence* of the pattern, which has to be checked. Let us call the fingerprint of the last q-gram of the pattern also the fingerprint of the pattern. In the BMH algorithm (Algorithm 4), which works with unigrams, $lastch$ is the fingerprint of the pattern and ch is a fingerprint computed from the text. This algorithm can be easily modified to handle larger values of q than 1. In Publication [I, p. 853] it is presented as Algorithm 2. There fingerprints are computed using Horner's rule: $t = a_0 + c(a_1 + c(a_2 + \cdots c(a_{q-2} + ca_{q-1}) \cdots))$.

Note that a straightforward implementation cannot process patterns shorter than q, but this problem is easy to avoid by incorporating Algorithm 4 to the implementation.

The shifts are taken according to the precomputed shift table D indexed by fingerprints. The computation of D is based on the definition:

$$D[x] = \min(m, \quad \min\{m - d \mid \\ 1 \le d < m, p_{d-e+1} \cdots p_d = \text{suffix}(x, e), e = \min(q, d)\}),$$

where $\text{suffix}(x, e)$ is the string of the last e characters of x. Baeza-Yates [Bae89] as well as Kim and Shawe-Taylor [KiS94] use a simpler alterna-

tive for the shift table D, which is faster to compute during preprocessing but which leads to slightly shorter shifts: $D'[x] = \min(m-q+1, \min\{m-d \mid q \leq d < m, p_{d-q+1} \cdots p_d = x\})$. We use mostly D in our algorithms.

At the implementation level, we use unrolled code instead of a loop for computing fingerprints, because a loop would make the algorithm slower. Kim and Shawe-Taylor [KiS94] use a loop where q is a constant. A compiler could unroll it to the straight code, and therefore the efficiency of this solution depends on the optimization skills of the compiler.

3.3 Alphabet transformation

The straightforward implementation of fingerprints contains a space problem, because the shift table D is indexed by the fingerprints. If the alphabet contains characters `a`, `c`, `g`, and `t`, whose ASCII codes are 97, 99, 103, and 116, respectively, the size of the actual alphabet is not four but 117 or 256. In order to handle the case $q = 4$, we might need a shift table of at least 117^4 elements. A large shift table also makes the preprocessing phase of the algorithm slow, because the table must be initialized. In practice, the initialization time of the table D with large values of q dominates the preprocessing time and even the total running time for short texts.

An alternative for multiplication in the computation of a fingerprint is to round the alphabet size c upwards to a power of two and to apply bitwise shifting [Bae89]. Bitwise shifting is considered faster than multiplication (of integers) in most computers [Knu11, p. 153], and the gain depends on the computer architecture. If c is not originally a power of two, this bitwise shifting approach requires a larger table D.

A third alternative is to replace multiplication by a table look-up. For example for $q = 4$ we have

$$a_0 + a_1 * c + a_2 * c^2 + a_3 * c^3 = a_0 + s[a_1 + s[a_2 + s[a_3]]],$$

where $s[a] = a \cdot c$. This method needs additional space and preprocessing time of order $\mathcal{O}(c^q)$ for the look-up table s.

One solution to the space problem is to apply hashing, which, however, makes the searching phase slower than table access. Zhu and Takaoka [ZhT87] present in addition a hashing scheme for $q = 2$. The size of the hash table is only twice the pattern length. In case of collision they use another hash function.

Another approach to save space is to cluster the alphabet by mapping input characters to a narrow range. The simplest mapping is to subtract 97 from the ASCII codes of characters `a`, `c`, `g`, and `t` to get 0, 3, 7, and 19, respectively, which means that the size of the resulting alphabet is 20. In Kim and Shawe-Taylor's [KiS94] mapping the last three bits of the character code are extracted, which means that the size of the resulting alphabet is eight. However, the initialization of the shift table is still laborious for large values of q in the alphabet of eight characters. Note also that the transformation of Kim and Shawe-Taylor does not work (without verification) for arbitrary character codes, because mapping may not be an injection. It would be also possible to to assign all other characters except `a`, `c`, `g`, and `t` to a code value of their own. Then the probability of false positives would decrease. Generally, if we know that certain characters do not occur in patterns, these characters can be safely mapped together.

Most of these simple approaches may produce errors, if the input happens to contain improper symbols. A safe way for mapping is to apply a direct alphabet transformation. We map the ASCII codes to the range of 4: $0 \leq r[j] \leq 3$ such that characters `a`, `c`, `g`, and `t` get different codes and possible other characters get e.g. code 0. In this way we are able to limit the computation to the effective alphabet of four characters. We use a separate transformation table h_i for each position i of a q-gram and incorporate multiplications into the tables: $h_i[j] = r[j] \cdot 4^i$. For $q = 4$, the fingerprint of $a_0 \ldots a_3$ is then computed as

$$h_0[a_0] + h_1[a_1] + h_2[a_2] + h_3[a_3].$$

It also possible to carry out the mapping of characters dynamically. Actually only characters appearing in the pattern matter: all other characters can be mapped to one code. This is often very efficient, because in the pattern there can exist at most m different characters.

Algorithm PP (shown as Algorithm 9) describes the preprocessing of the transformation tables and the shift table D for $q = 4$. The algorithm can be implemented so that there is a different code for each value of q (q can be a compilation parameter in C). Note that equations $r[j] = h_0[j]$ and $h_3[x] = h_0[x] \cdot c^3$ hold.

Note that when the data contains more than four different characters, the mapping r is not an injection. So this mapping defines a simple hashing scheme. In algorithms using fingerprints with injective mapping, the q-grams never needs to be re-examined in the checking phase. In the case

Algorithm 9 PP($P = p_1 p_2 \cdots p_m$)

/* Initializing transformation tables */
1: $c \leftarrow 4; \quad q \leftarrow 4$
2: **for all** $i \in \Sigma$ **do**
3: $\quad h0[i] \leftarrow 0$
4: $h0['a'] \leftarrow 0; \quad h0['c'] \leftarrow 1; \quad h0['g'] \leftarrow 2; \quad h0['t'] \leftarrow 3$
5: **for all** $i \in \Sigma$ **do**
6: $\quad h1[i] \leftarrow c \cdot h0[i]; \quad h2[i] \leftarrow c \cdot h1[i]; \quad h3[i] \leftarrow c \cdot h2[i]$
/* Initializing shift table D */
7: $u \leftarrow c^q$
8: **for** $i \leftarrow 0$ to $u - 1$ **do**
9: $\quad D[i] \leftarrow m$
10: $s \leftarrow 0; \quad a \leftarrow u$
11: **for** $i \leftarrow 1$ to $q - 1$ **do**
12: $\quad s \leftarrow s$ div $c + h3[p_i]$
13: $\quad a \leftarrow a/c$
14: $\quad$ **for** $j \leftarrow s$ to $s + a - 1$ **do**
15: $\quad\quad D[j] \leftarrow m - i$
16: $s \leftarrow s$ div $c + h3[p_q]$
17: **for** $i \leftarrow q + 1$ to m **do**
18: $\quad D[s] \leftarrow m - i + 1$
19: $\quad s \leftarrow s$ div $c + h3[p_i]$

of a non-injective mapping the q-gram must be re-examined, if the fingerprints are equal and $p_1 p_2 \cdots p_{m-q}$ match with the text. However, the slowdown due to this is insignificant in the average case when pairwise comparison is made forward.

When the data contains more than four different characters, there is also another problem. Namely, if a q-gram of the text contains a rare character, the shift for it is possibly shorter with the mapping r than without it. For example, if the pattern is `cacgtcccc` and the q-gram is `xcgt` and r[a] = r[x], the shift for this q-gram is 5 with the mapping and 9 without it. Nevertheless, the frequency of other characters than `a`, `c`, `g`, and `t` is very low in real DNA data so that this slow down is insignificant in practice.

When the character transformation is applied to a larger alphabet, where there are several text characters that do not occur in the pattern, all these characters can be mapped together. If the total frequency of these characters is low, the size of the effective alphabet is the number of

different characters in the pattern, otherwise one more.

An example of applying alphabet transformation in fingerprints is Algorithm 3 in Publication [I, p. 856].

A perfect hash function for a set S is a hash function that maps distinct elements in S to a set of integers, with no collisions. A minimal perfect hash function is a perfect hash function that maps r keys to r consecutive integers. For ASCII characters `a`, `c`, `g`, and `t`, a minimal perfect hash function is in the C language

```
inline int symtocode2(int sym)
                {return (((sym+sym+sym) & 0x18) >> 3);}
```

This is slightly more laborious than the approach by Kim and Shawe-Taylor.

3.4 Applying skip loop

The BMH algorithm makes at least two tests at every alignment: has the pattern reached the end of the text and does the last character of the pattern match the last text character in current alignment? The situation is similar while using fingerprints. The main loop would be faster, if one of the tests could be removed. The unrolled fast skip loop is ideal for this situation. During preparation of the Publication [I] even higher unrolling factors than 3 seemed competitive in the tests. In the Publication [IV] a newer and different kind of computer was used, and the unrolling factor 1 was found the best on DNA and amino acid texts. Increasing the fingerprint size typically increases the probability that the algorithms do not fall to the slow loop.

Algorithm 10 (called BMHq) implements fingerprint method with unrolling factor 1.

In Algorithm 10, $f(T, k, q)$ denotes the fingerprint of $t_{k-q+1} \cdots t_k$. The pattern needs to be copied beyond the end of the text, so that the unrolled fast skip loop will end when the search is complete. Initialization of the shift table D similar as in Algorithm PP (Algorithm 9) is slower than just using full q-grams existing in the pattern. Then the maximal shift length is $m - q + 1$ instead of m. This has practical consequences when q gets larger or when the pattern is relatively short.

In Publication [I] the Algorithm 10 with the unrolling factor 3 was called A4.4. A potential explanation to the difference in the relative speed

Algorithm 10 BMHq($P = p_1p_2\cdots p_m, T = t_1t_2\cdots t_n$)

Require: $n \geq m \geq q > 0$

```
 1: Initialize D[*]                          /* executing e.g. Algorithm PP */
 2: t_{n+1}...t_{n+m} ← P                                  /* adding stopper */
 3: p ← f(P, m, q)
 4: r ← D[p];   D[p] ← 0;   k ← m
 5: s ← D[f(T, k, q)]
 6: loop
 7:     while s > 0 do
 8:         k ← k + s
 9:         s ← D[f(T, k, q)]
10:     if k > n then exit
11:     Check the potential occurrence
12:     s ← r
```

of A4.4 and KS algorithm (by Kim and Shawe-Taylor) in Publications [I] and [IV] is the change in the value of the optimal unrolling factor. However, the effect of unrolling is rather small for long DNA patterns.

3.5 Simultaneous read of several bytes

Some CPU architectures, notably the x86, allow unaligned memory reads of several bytes. This inspired us to try reading several bytes in one instruction, instead of separate character reads. One may argue that it is not fair to apply such multiple reading, because all CPU architectures do not support it. But because of the dominance of the x86 architecture, it is reasonable to fine-tune algorithms for that. Of course the results may be different on other platforms.

Fredriksson [Fre03] was probably the first one who applied reading several bytes simultaneously to string matching. However, we have not seen any other comparison with standard byte wise reading than in our article [PeT11].

When considering 4-grams, they fit into a 32-bit word in 8-bit characters. This inspired us to try reading several bytes in one instruction, instead of four separate character reads. During the preprocessing phase, we have to take care of endianess (the order in which integer values are stored as bytes in the computer memory). Also the fingerprint from the pattern must be formed accordingly.

Reading more than two bytes simultaneously does not seem to give extra advantage. Based on the tests in Publication [IV], unaligned memory reads on x86 processors incur a speed penalty of up to 70% when compared with aligned reads. This unfortunately reduces the speed of reading four bytes, because then 75% of the reads are unaligned on the average.

3.6 Variations of fingerprint method

3.6.1 Variations for DNA

As in Publication [IV] we will call the BMHq algorithm with $q = 4$ BMH4. We tested several ways to compute the fingerprint from DNA alphabet in order to make faster versions of the BMH4 algorithm. BMH4b reads a 32-bit word, and BMH4c reads two consecutive halfwords. Because in BMH4b we have access to an integer consisting of four characters, it would be inefficient to use the old character-based fingerprint method. The fingerprint calculation arrays of BMH4 are replaced with hashing expression, where the input is a whole 4-gram as a 4-character long integer. ASCII codes for `a`, `c`, `g` and `t` are distinguishable by the last three bits. The following expression packs the unique bits of the four characters together in a few instructions, to form an integer in the range of $2313 \ldots 16191 = 00100100001001_2 \ldots 11111100111111_2$.

```
FP(x) = ((x >> 13) & 0x3838) | (x & 0x0707)
```

Preprocessing of BMH4b is similar to the earlier algorithm, we just calculate the fingerprints of the 4-grams in the pattern with the new hash function. So the main difference between BMH4b and BMH4 is in the computation of f. In BMH4b this is done with masking, shifting, and bitwise OR. Hashing receives the current text location pointer as an argument, and reads the 32-bit integer from that address with `FP(*(k-3))`. $D[x]$ contains the preprocessed shift values for each hashed q-gram of the pattern.

75% of the reads are unaligned on the average. So we made another variation BMH4c, which reads two consecutive halfwords. In the case of BMH4c, only 25% of the reads are unaligned ones getting the speed penalty (while crossing the border of 4 bytes). In BMH4c, the value of a fingerprint is obtained as $a_1[x_1] + a_2[x_2]$ where x_i is a halfword and a_i a preprocessed transformation table, for $i = 1, 2$.

3.6.2 Variations for Amino Acids

The q-gram approach is valid also for larger alphabets, although with a larger alphabet the optimal value of q is smaller. A larger alphabet size requires only minor changes to the algorithm. A new variation BMH2 was created based on BMHq with $q = 2$. The range of the ASCII code mapping $r[x]$ is increased from 4 to 20, to cover the amino acid alphabet "ACDEF GHIKL MNPQR STVWY" instead of the DNA alphabet. Otherwise the algorithm is the same as BMHq.

We also designed BMH2c, which reads a 2-gram as a 16-bit halfword. The shift array is indexed directly with halfwords.

These algorithms can be used with any text, e.g. English text. For this kind of data we mapped each character to a smaller range with a modulo function in preprocessing. The best results for English text were obtained with modulo 25. Because the mapping tables are created in the preprocessing phase, the modulo operation does not affect the search time directly.

3.7 Lecroq's hashing scheme

Recently, Lecroq [Lec07] presented a related algorithm using hashing. Its implementation is based on the Wu–Manber algorithm [WuM94] for multiple string matching, but as suggested above, the idea is older [BoM77, ZhT87]. Originally, the algorithm was called 'New', but lately a more descriptive name 'Hash$q$' was used [FaL10]. The hashing scheme is based on computing fingerprints of q-grams. Lecroq's Hash$q$ algorithm is closely related to BMHq. Moreover, the maximal shift of his algorithm is $m-q+1$, while that of BMHq is m, because BMHq is able to handle all prefixes of the first q-gram of the pattern. With $q = 3$, the shift in HASHq is computed from the hashvalue in the following way:

12: $\quad h \leftarrow t_{j-2};\ h \leftarrow 2 \cdot h + t_{j-1};\ h \leftarrow 2 \cdot h + t_j$

13: $\quad sh \leftarrow D[h \bmod 256]$

In the original article, only divisor 256 is mentioned. Lecroq's hashing scheme is extremely simple and fast to evaluate. However, collisions are quite probable. DNA alphabet consists mostly of characters a, c, g, and t. Their decimal values are 97, 99, 103, and 117, respectively. 2-grams 'ag'

and ‘cc’ produce the same value:

$$\begin{aligned} 97 \cdot 2 + 103 &= 297 \\ 99 \cdot 2 + 99 &= 297 \end{aligned}$$

One of the consequences is that with the DNA alphabet about 1/8 of the q-grams hash to same value. Also 3-grams ‘agc’, ‘cag’ and ‘ccc’ hash to the same value:

$$\begin{aligned} 97 \cdot 4 + 103 \cdot 2 + 99 &= 693 \\ 99 \cdot 4 + 97 \cdot 2 + 103 &= 693 \\ 99 \cdot 4 + 99 \cdot 2 + 99 &= 693 \end{aligned}$$

With the alphabet {a, c, g, t}, there are 4^q q-grams, and at most $m - q + 1$ different of these are in the pattern. Table 3.1 lists the number of collisions on various values of q and various divisors. It is very probable that with larger alphabets more collisions might occur. If the divisor operand of modulo is small, the expected shift length will increase only slightly after a certain pattern length. This happens gradually when the number of possible q-grams exceeds the hash table size. The results clearly suggest that even on a small alphabet a larger divisor than the original 256 would work clearly better when $q > 4$. The Hash$q$ algorithm would work

Table 3.1. Collisions with the alphabet {a, c, g, t} in the Lecroq’s hashing scheme. Percentages of collisions in the second column are counted without any divisor i.e. the final value of h was the same.

q	percentage of collisions	divisor 256		divisor 1024		divisor 4096	
		unique elements	unused elements	unique elements	unused elements	unique elements	unused elements
3		43	203	43	971	43	4043
	32.81	16.80%	79.30%	4.199%	94.82%	1.050%	98.70%
4		90	100	94	856	94	3937
	63.28	35.15%	39.06%	9.180%	84.47%	2.295%	96.12%
5		33	25	165	609	165	3681
	83.89	12.89%	9.766%	16.11%	59.47%	4.028%	89.87%
6		10	4	173	119	237	3122
	94.21	3.906%	1.562%	16.89%	11.62%	5.786%	76.22%
7		2	1	33	25	309	1955
	98.11	0.781%	0.391%	3.223%	2.441%	7.544%	47.72%
8		1	0	10	4	175	120
	99.42	0.391%	0.000%	0.977%	0.391%	4.272%	2.930%

with reasonable preprocessing time even without any division when $q \leq 9$. The experimental results given in the original article [Lec07] suggest that larger divisor, than 256 have been used at least on the longest patterns. In a pattern there can be at most $m-q+1$ different q-grams. In the binary alphabet there are at most 2^q different q-grams. When $m-q+1 \gg 2^q$ there will be relatively more collisions. The practical performance will increase only little, when the pattern length increases. This phenomenon can be seen in Table 5 [Lec07, p. 233]. On the other hand, the relative number of collisions increases, when $m - q + 1$ exceeds the hash table size. In Table 6 [Lec07, p. 233] using DNA patterns there seems to be no slowdown with values $q \geq 5$. With this small hash table there should happen a minimal number of cache misses in the data cache.

Of course the divisor in hashing could be different from 256. The use of 256 as the divisor is practical since the remainder can be stored in one byte. For example, in the programming language C, the type of the variable h can be `unsigned char`. And the code optimizer removes the unnecessary modulo operation; this happens at least with the gcc compiler. Also, masking the lower bits from h seems to be faster than the modulo operation. The optimizer of gcc is able to do that when the divisor is a power of 2. So, divisors that are powers of 2 seem to be the best in practice.

4. Bit-parallel string matching

The register width of a processor has recently quite commonly grown from 32 to 64 bits. The relative number of 64 bit computers has lately increased. Therefore, the bit-parallel algorithms are in practice able to work efficiently on longer patterns than before.

In the x86 and x86-64 architecture MMX brought registers having 64 bit, SSE (Streaming SIMD Extensions) offers 128-bit registers, and in AVX (Advanced Vector Extensions) register size increased to 256 bits. Generally, these special registers are used for operating several smaller size data types. For example a 128-bit SSE register can work on sixteen 8-bit bytes or four 32-bit integers. In essence, we can regard the subfields of a register as if they were elements of an array of independent microprocessors, acting independently on their own subproblems yet tightly synchronized, and communicating with each other via shift instructions and carry bits [Knu11, p. 151]. However, the width of the memory bus may then become a bottle neck resource in the application of bit-parallel algorithms.

Our first efforts in developing bit-parallel algorithms took their form during the preparation of Publication [II]. Especially the usefulness of simplification became clear. The relatively high number of conditional branches in the BNDM algorithm seemed to be questionable for the performance. However, during the same time the branch prediction of processors developed quickly. After we got the source code of SBNDM2, the original version of still more simplified version of BNDM[1], we quickly produced versions reading more and more text characters at the beginning of each alignment. However, the best number of text characters on various pattern lengths was often unexpected in test runs for Publication [V].

[1]Jorma Tarhio noticed a shift inefficiency in the first version of SBNDM2 [HoĎ05].

The notations used here were introduced in Section 2.5.

4.1 Shift-Vector Matching

A problem with the BNDM algorithm, the Boyer–Moore algorithms, and the most related algorithms related to them, is that they do not remember from previous alignments which text positions cannot start a match. When the shift is shorter than the pattern length m, some alignments of the pattern may be checked in vain. In this section, we introduce an algorithm with partial memory. The key idea is simple: We keep track of end positions corresponding to those start positions mentioned above. We maintain a bit-vector, called a *shift-vector*, which keeps track of the positions where an occurrence of the pattern can or cannot end. When shifting is based on this shift-vector, we are able to manage without any shift table.

While moving the pattern forward and shifting the shift-vector, the old knowledge of already handled positions drops off the shift-vector. Then the bit corresponding to the end of the pattern must be the highest or the lowest bit. We chose the lowest one, because it makes masking on some processors slightly faster. (Often the fastest way to load the specific bit mask 1 to a register is loading the constant 1 with some instruction, which is not referring to the memory.) This decision implies that the shifting direction is to the right. The new bits entering to a bit-vector during a bitwise shift are zeros, and therefore it is natural to use the convention where zero denotes a text position not yet rejected.

In preprocessing, a bit-vector is created for each character of the alphabet. These bit-vectors have the zero bit on every position where that character occurs in the pattern and one elsewhere. So the characters that do not appear in the pattern have the bit-vector $0^{w-m}1^m_2$. Note that the essential parts of these vectors are complements of those used in BNDM.

We keep track of possible end positions of the pattern in the shift-vector S. It is simply updated by taking OR with the bit-vector corresponding to text character aligned with the last character of the pattern. If the lowest bit in S is one, a match cannot end at the corresponding position and we can shift the pattern. The length of the shift is simply obtained by searching the lowest zero bit in S which is above the lowest position. In addition to shifting the pattern, we also shift bits in S with the same number of positions to the right.

If the lowest bit in S is zero, i.e. p_m has been found, we have to continue checking for the match. Our first implementation performed a classical pairwise comparison of pattern and text characters. In addition, S was updated with all characters that were fetched during verifying of alignments. Naturally the scope of text characters is relative to the end of pattern. To correctly update S with bit-vectors of text characters, that are aligned with pattern, their values have to be shifted to the right depending on how far they are from the end of the pattern: $S \leftarrow S \mid (C[t_{\mathit{epos}-j}] >> j)$. Because the lowest bit remains zero as long as a mismatch has not been found, we could remove the pairwise comparison. Text characters that are on the left-hand side of the alignment give less information for shifting than those which are close to the right end of the alignment. That is why we chose to check, if there is a match, in the reverse order, i.e. from right to the left.

The pseudo-code of SVM (short for Shift-Vector Matching) for $m \leq w$ is shown as Algorithm 11. The function BSF (short for Bit Scan Forward)

Algorithm 11 SVM($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $m \leq w$

/* Preprocessing */

1: **for all** $c \in \Sigma$ **do** $C[c] \leftarrow (\sim 0) >> (w - m)$ /* 1_2^m */

2: **for** $j \leftarrow 1$ **to** m **do**

3: $\quad C[p_j] \leftarrow C[p_j]$ & $\sim(1 << (m - j))$ /* $1^{j-1}01_2^{m-j}$ */

/* Searching */

4: $\mathit{epos} \leftarrow m; \quad S \leftarrow 0$

5: **while** $\mathit{epos} \leq n$ **do**

6: $\quad S \leftarrow S \mid C[t_{\mathit{epos}}]$

7: $\quad j \leftarrow 1$

8: $\quad$ **while** $(S\&1) = 0$ **do**

9: $\quad\quad$ **if** $j \geq m$ **then**

10: $\quad\quad\quad$ report an occurrence at $\mathit{epos} + 1 - m$

11: $\quad\quad\quad$ **goto** Over

12: $\quad\quad S \leftarrow S \mid (C[t_{\mathit{epos}-j}] >> j)$

13: $\quad\quad j \leftarrow j + 1$

14: $\quad$ Over:

15: $\quad \mathit{last} \leftarrow$ BSF($\sim(S >> 1)$)+1

16: $\quad S \leftarrow S >> \mathit{last}$ /* if $m = w$, see implementation remark */

17: $\quad \mathit{epos} \leftarrow \mathit{epos} + \mathit{last}$

scans the bits in the operand bit-vector starting from the lowest bit and searches for the first set bit. The function returns the number of zero bits before the first set bit. At the end of this section, we discuss various implementation alternatives of BSF.

We demonstrate the execution of SVM with an example in Table 4.1. After a long shift, the shift-vector S becomes zero or almost zero. Then the subsequent shift is more likely shorter. Fortunately after a short shift, there will normally be several ones in S, and so the subsequent shift will likely be longer again. For example, after reading G on the second row of the example, S becomes full of ones enabling a full shift of 5 positions.

In SVM, there is an obvious trade-off between the number of fetched characters and searching for a set bit in the shift-vector. The run times depend on the relative speed of these functions. It is straightforward to extend this algorithm for longer patterns.

Table 4.1. Simulation of SVM. P = ATCGA; T = GCAGCTATCGAG$\cdots$; bit-vectors C: C[A] = 01110, C[C] = 11011, C[G] = 11101, C[T] = 10111. The last fetched character has been underlined. The snapshots correspond to lines 9 and 15 of the SVM algorithm.

Text	S	j	*epos*	*last*
GCAG<u>C</u>TGATCGAG$\cdots$	11011	1	5	2
GCAGCT<u>G</u>ATCGAG$\cdots$	11111	1	7	5
GCAGCTGATCG<u>A</u>G$\cdots$	01110	1	12	(5)
GCAGCTGATC<u>G</u>AG$\cdots$	01110	2	12	(5)
GCAGCTGAT<u>C</u>GAG$\cdots$	01110	3	12	(5)
GCAGCTGA<u>T</u>CGAG$\cdots$	01110	4	12	(5)
GCAGCTG<u>A</u>TCGAG$\cdots$	01110	5	12	5

Navarro and Raffinot [NaR00, p. 14] also use the method of searching for a certain bit in a bit-vector describing the state of the search. However, they consider only one alignment at a time and they initialize the bit-vector for each alignment. In SVM, we initialize the bit-vector only once and so we are able to exchange information between alignments. SVM searches only for complete matches and does not recognize substrings of the pattern like BNDM.

The weakness of SVM is the laborious shifting of the pattern. For SVM it is easy to utilize text characters that are ahead from current alignment. All information can be incorporated to shift-vector S from the characters that are at most $w - m$ forward from the end of the alignment. Then

it is possible to make longer shifts. We have made some tests with this approach. Shifts became longer, but the performance improved only a little.

Complexity. Let us assume that $m \leq w$ holds and BSF is $\mathcal{O}(1)$. Then the preprocessing time is $\mathcal{O}(m + |\Sigma|)$ and the worst case complexity is clearly $\mathcal{O}(nm)$. SVM is sublinear on the average, because at a given alignment it fetches the same text characters as the Boyer–Moore–Horspool algorithm [Hor80] (assuming the right-to-left examining order) and can never make a shorter shift than that algorithm, which is known to be sublinear on the average. If a constant time BSF is not available, there will be an extra work of $\log m$ or $\log w$ for each alignment.

Search for the lowest zero bit

From the previously examined characters we usually know some positions where the pattern cannot end. All these positions with reference to the end of the pattern have the corresponding bit set in the shift-vector S of SVM. The lowest bit represents the current position. To get the length of the next shift of the pattern, one has to find the rightmost zero bit in S. Alternatively, one can complement the bits and search for the lowest set bit. This problem is often also called the *number of trailing zeros*.

There are several possibilities [War03, p. 6, pp. 84–87] for searching the rightmost set bit. Presumably, many solutions have not been documented and published. Below, we consider five alternatives: BSF-0, ..., BSF-4. If we first shift the contents of the word one position to the right and if we are using unsigned variables in C, we get zero padding and there will always exist at least one zero bit. This simplifies BSF and ensures that the shift length in shift-vector S is less than w.

BSF-0. Many computer architectures have instructions for scanning bits; for example Intel's x86 has instructions for scanning both forward (*Bit Scan Forward*) and backward (*Bit Scan Reverse*)[2]. A suitable implementation can be found in Arndt's collection [Arndt] of x86 inline `asm`[3] versions of various functions as function `asm_bsf`. For 32-bit bit-vectors, the *bsfl* instruction is suitable, and in the 64-bit architecture, there is the *bsfq* instruction for 64-bit bit-vectors.

[2]AMD's Barcelona microarchitecture offers the SSE4a instruction group. There is the LZCNT – Leading zero count – instruction, which work similarly as BSR – Bit Scan Reverse.

[3]http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

Bit Scan Reverse instruction actually returns "integer logarithm" from an unsigned integer, which means $\lfloor \log_2 x \rfloor$. Standard function `ilogb` in the programming language C (starting year 1999 standard) returns also this same value. It is also highly portable. To use these in SVM, we should reverse the shift-vector S and the bit-vectors of text characters in $C[\,]$.

If this kind of machine instruction is available, using it likely gives the best performance.

BSF-1. The simplest way to seek for the lowest zero bit is by shifting the bit-vector bit position by bit position to the right and testing the lowest bit. If the lowest zero bit can be found with a few iterations—e.g. when m is small—the performance is acceptable. Navarro and Raffinot [NaR00] have used this technique in their implementation of BM_BNDM and TurboBNDM. Warren [War03, code 5-16, p. 86] presents a similar code. Starting from the m^{th} text character BSF-1 tests every position either once or twice. Therefore over the whole text, the total work required by BSF-1 is $\Theta(n)$. The relative performance decreases while the average length of shift increases.

Every now and then one finds the usages of `likely()` or `unlikely()`. They resemble function calls but are actually macro definitions for the `gcc` compiler. They hint[4] which alternative of the boolean expression given as a parameter is more probable. Both `likely()` and `unlikely()` were tried in the alternative implementations of BSF-1. Test results on the Pentium D processor seemed to have a negative effect in most cases.

BSF-2. Search for the lowest set bit becomes easier, if we assume that at most one bit is set. This can be achieved with an expression $x \,\&\, -x$. The rightmost, i.e. least significant set bit remains set [War03, p. 11]. This process assumes that signed integers are in the two's complement representation.[5]

If at most one bit is set, it is possible to apply bit masking: we divide different sized groups in the bit-vector to an upper and lower half. If

[4]The Pentium 4 Processor introduced new instructions for adding static hints to conditional branches: it informs whether or not a branch is assumed to be taken. In static branch prediction, a forward branch defaults to 'not taken', in other words, processor assumes that the jump is not made; whereas a backward branch defaults to 'taken'.

[5]In the C language this can always be achieved by using 'Exact-width integer types', e.g. define bit-vectors to be of type `uint32_t` or `uint64_t`. Then the corresponding signed types are `int32_t`, and `int64_t`, which must always have the two's complement representation.

some even bit is set, then we can increase the bit number by one. If some even bit-pair is set, then we can increase the bit number by two. If some upper half of byte is set, then we can increase the bit number by four; etc. Finding the set bit this way requires $\log w$ tests with different bit masks. This idea is presented in the function `lowest_bit_idx` of Arndt [Arndt].

The masking method described above requires large (i.e. w bits wide) bit masks. If they are built from smaller pieces, their construction takes considerable work.

BSF-3. This approach is similar to the previous one. Here we clear all other but the lowest set bit. If an unsigned integer after shifting l positions to the right is not zero, it is obvious that the only set bit is higher than l bits. The search goes most efficiently by halving: first $w/2$ bits, then $w/4$ bits, etc. The shifting could be made also to the left, but in this way the optimizer of the compiler can produce more efficient code by reusing the results of the shifts. Examining the last byte goes faster and easier with table lookup in a precomputed constant array. Altogether $\log \frac{w}{8}$ shifting tests are needed. Actually, the same holds also for $m \lceil \log \frac{m}{8} \rceil$ because one can tailor the routine for different pattern lengths. Relative performance improves clearly when patterns get longer. Warren [War03, code 5-6, p. 78] presents a similar shifting code for the number of leading zeros, but without a precomputed array.

Branch optimizations have a significant impact on performance. The first part of BSF-3 can be implemented in C with 32 bit bit-vectors in the following way. `x` is a bit-vector having at most one set bit, and `j` gets the offset to the byte having the set bit.

```
j = 0;
if(x >> 16) {j += 16; x >>= 16;}
if(x >>  8) {j += 8;  x >>= 8;}
```

This can also be implemented without any conditional or unconditional jump instructions:

```
j  = (((x >> 16)==0)-1)&16;  /* j gets 0 or 16 */
j += (((x >> j+8)==0)-1)&8;
```

It turned out that the latter version works slower than the former version.

BSF-4. We can also utilize the fact that at least one bit is set. The basic idea is that when we shift to the left and the result is zero, we can conclude that the lowest set bit was in the part that fell off. Because we try

to isolate the lowest bit to smaller and smaller area, for the next step we have to shift the bit-vector to the right whenever the result is zero after shifting. Examining the last byte is made with a table lookup in a precomputed constant array. A similar code without a precomputed array is presented by Warren [War03, figure 5–14, page 85].

Typically, SVM takes a couple of shorter shifts and then a longer one, usually the maximum m. In our experimental tests, we used the version BSF-0 utilizing the inline `asm` function.

Implementation remark. When $m = w$, the value of *last* may become w, which is too large for the shift on line 16. Then the shifting must be made in two parts. This can be made efficiently with the following changes:

14: Over: $S \leftarrow S >> 1$

15: $last \leftarrow \text{BSF}(\sim S)$

16: $S \leftarrow S >> last$

17: $epos \leftarrow epos + last + 1$

The performance of all the BSF versions depends a lot on the compiler, the computer architecture, and the size of bit-vectors. Besides AMD Athlon, we tested BSF on the following configurations: Sun Enterprise 450 (4 UltraSPARC-II 400MHz processors, 2048 MB main memory with 4 MB Ecache, Solaris 8) with Sun WorkShop 6 update 2 C 5.3 and gcc 3.2.1 compilers, and Digital Personal Workstation 433au (Alpha 21164A-2 (EV56 433 MHz) processor, 256 MB main memory; OSF1 V5.1 [Tru64 UNIX]) with Compaq C V6.5-011 and gcc 3.3 compilers.

Table 4.2 shows the relative performance of SVM with various BSF versions on the English text, where BSF-4 is used as the reference version (so its relative performance is 1). Smaller values denote faster performance.

Table 4.2. Relative performance of various BSF versions

CPU	AMD Athlon		UltraSPARC-II				Alpha 21164A-2	
Compiler/w	gcc/32	gcc/64	Sun/32	gcc/32	Sun/64	gcc/64	Compaq(64)	gcc(64)
BSF-0	0.888	–	–	–	–	–	–	–
BSF-1	1.001	0.817	1.346	1.191	1.146	0.972	0.730	1.036
BSF-2	1.044	1.157	1.420	1.126	1.211	1.348	1.011	1.217
BSF-3	0.999	0.999	0.992	0.999	1.123	1.109	1.005	1.001

With longer patterns, the relative performance of BSF-1 got worse. Although Ultra-Sparc-II has 64-bit instructions, the use of bit-vectors of 64 bits showed to be more than 60% slower than with 32 bits. The Alpha Compaq C-compiler produced code that worked rather slowly with BSF-2, BSF-3, and BSF-4.

Because the maximum value of BSF is the pattern length m, we could speed up the BSF-functions by using different specialized versions for $m \leq 8$, $8 < m \leq 16$, $16 < m \leq 32$, and $32 < m \leq 64$.

4.2 Two-way variant of BNDM

Next we introduce a two-way modification of the BNDM algorithm. If the text character aligned with the end of the pattern is a mismatch, we continue by examining text characters after the alignment. Let us consider TNDM (short for Two-way Nondeterministic DAWG Matching) in detail. Furthermore let us consider the first comparison of an alignment of the pattern: t_i vs. p_m. There are three cases:

1. $t_i = p_m$;
2. $t_i \neq p_m$ and t_i occurs elsewhere in P, i.e. there exists $j \neq m$ such that $t_i = p_j$;
3. t_i does not occur in P.

The TNDM algorithm works as BNDM in Cases 1 and 3, but the operation is different in Case 2, where the standard BNDM continues examining backwards until it finds a substring that does not occur in the pattern or it reaches the beginning of the pattern. TNDM will scan *forward* in Case 2. Our aim is to reduce the number of examined characters. In a way, this approach is related to Sunday's idea [Sun90] for using the text position immediately to the right of an alignment for determining the shift in the Boyer–Moore algorithm.

The cache memories constitute a very important part of the current computers. The data from adjacent bytes in memory is mapped to cache in blocks called a cache line. Their current typical size is 32 or 64 bytes. When any byte is fetched from the memory, the whole corresponding cache line is fetched to the cache. To reduce the latency caused by cache misses, various hardware prefetch mechanisms are used. A common one is ad-

jacent cache line prefetch[6]. When it is in use, the content of next cache line is also fetched to the data cache. Therefore when a program has read a byte from a text position also some amount of bytes after it also are fetched to the data cache ready to be read. So the text characters immediately to the right of an alignment are with high probability already in the cache memory. Thus, the expected time for fetching them is almost always minimal.

In Case 2, the next text characters that are fetched are not needed for checking a potential match in the BNDM algorithm, but they are only used for computing the shift. Because $t_i \neq p_m$ holds, we know that there will be a shift forward anyway before the next occurrence is found. The idea of TNDM is to examine text characters forward one by one until the algorithm finds the first k such that the string $t_i \cdots t_k$ does not appear in P or $t_i \cdots t_k$ forms a suffix of P. In the former case, we can shift beyond the previous alignment of the pattern.

Checking whether the examined characters form a suffix of the pattern, is made by building the identical bit-vector as in BNDM, but in the reverse order. Note, that the bit-vector is built with the (logical) AND operations which are commutative. So we can build it in any order—especially in the reverse order. Instead of shifting the bit-vector describing the state, we shift the bit-vectors of characters. Thus, if we find a suffix, we continue to examine backwards starting from the text position $i - 1$. This is done by resuming the standard BNDM operation.

The pseudo-code of TNDM for $m \leq w$ is shown as Algorithm 12. It is straightforward to extend the algorithm for longer patterns in the same way as BNDM, see [NaR00]. Because BNDM is a bit-parallel implementation of BDM, it would also be possible to make a two-way modification of BDM.

To be able to resume efficiently examining backwards, i.e. jumping in the middle of the main loop of BNDM, we preprocess the possible values of the variable *last* of BNDM for the suffixes of the pattern. With *last*, BNDM keeps track of the starting position of the next potential occurrence P. By updating the state vector in a clever way during the forward phase, we keep it ready for the backward phase.

In preprocessing, the values of *last* are computed with the BNDM al-

[6]On processors based on the Intel NetBurst microarchitecture this feature is enabled through the BIOS. URL: http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers.

Algorithm 12 TNDM($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $m \leq w$

```
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 0                                   /* 0_2^m */
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j))                        /* 0^{j−1}10_2^{m−j} */
 4: Init_shift(P, restore[ ])
    /* Searching */
 5: epos ← m
 6: while epos ≤ n do
 7:     i ← 0;   last ← m
 8:     D ← B[t_epos]
 9:     if (D&1) = 0 then /* when D ≠ B[p_m], */
10:         repeat /* forward scan for suffix of pattern */
11:             i ← i + 1
12:             D ← D & (B[t_{epos+i}] << i)
13:         until D = 0_2^m and D&10_2^i = 0_2^m
14:         if D = 0_2^m then /* already last ← m */
15:             goto Over
16:         epos ← epos + i;   last ← restore[i]
17:     repeat /* variation of BNDM */
18:         i ← i + 1
19:         if D&10_2^{m−1} ≠ 0_2^m then
20:             if i < m then last ← m − i
21:             else report an occurrence at epos − m + 1; goto Over
22:         D ← (D << 1) & B[t_{epos−i}]
23:     until D ≠ 0^m
24:     Over:
25:     epos ← epos + last
```

gorithm as if there was a full occurrence of the pattern in the text. Algorithm 13 shows the pseudo-code where the values of *last* are stored in the array *restore*[7]. We demonstrate the execution of TNDM with an example in Table 4.3.

Our experiments indicate that TNDM examines fewer characters than BNDM on the average. There are two reasons for that. Let $t_i \cdots t_k$ be the string examined during the forward phase.

[7] In the Publication [II] there were two errors on line 6.

Algorithm 13 Init_shift($P = p_1p_2 \cdots p_m$, *restore*[ ])

```
1: D ← (~0) >> (w − m)                                   /* 1_2^m */
2: last ← m
3: for i ← m downto 1 do
4:     D ← D&B[p_i]
5:     if D&10_2^{m−1} ≠ 0_2^m then /* D&(1 << (m − 1)) ≠ 0 */
6:         if i > 1 then last ← i − 1
7:     restore[m − i + 1] ← last
8:     D ← D << 1
```

- When $t_i \cdots t_k$ is a suffix of P, we shift the pattern to that suffix. The suffix need not to be reexamined for a possible match ending at t_k. (If BNDM finds a prefix $t_h \cdots t_i$, that prefix may be reexamined for a possible match starting at t_h.)

- If $p_1 \neq p_m$ and $t_i = p_1$ hold, TNDM may make a shift one position longer than BNDM.

It is not difficult to find examples where TNDM examines more characters than BNDM. However, there is always a dual case where the situation is the other way around. Basically, BNDM searches for a substring $t_h \cdots t_i$ and TNDM for a substring $t_i \cdots t_k$ which do not appear in P. Which one is more efficient depends on the ratio $(k - i)/(i - h)$.

Further enhancements. If the last character examined does not occur in P while scanning forward, we are able to shift the pattern entirely over it. This can be done by replacing the following line to TNDM:

```
15:         if B[t_{pos+i}] = 0 then last ← i + m; goto Over
```

This test is computationally light, because after a forward scan only t_k of $t_i \cdots t_k$ can be missing from the pattern. The test clearly reduces the number of fetched characters. However, the test is beneficial only for alphabets large enough.

In TNDM we scan forward when t_i is not p_m and t_i occurs elsewhere in P. This can be generalized as follows. If the backward phase has encountered $v = t_h \cdots t_i$ such that v is not a suffix of P but v appears elsewhere in P, we will scan forward starting from t_{i+1}. We expect this modification would improve TNDM a bit in the case of small alphabets.

Table 4.3. Simulation of TNDM. P = ATCGA; T = GCATCATGATCGAATCAG$\cdots$; bit-vectors B: B[A] = 10001, B[C] = 00100, B[G] = 00010, B[T] = 01000. The last fetched character has been underlined.

Text window	Line	D	i	*epos*	*last*	Explanation
GCAT<u>C</u>ATGA$\cdots$	9	00100	0	5	5	The lowest bit is 0; continue to line 10.
GCATC<u>A</u>TGA$\cdots$	13	00000	1	5	5	$D = 0$; leave the loop and proceed with lines 14, 15, 24, 25, and 6–.
$\cdots$ATGA<u>T</u>CGAA$\cdots$	9	01000	0	10	5	The lowest bit is 0; continue to line 10.
$\cdots$ATGAT<u>C</u>GAA$\cdots$	13	01000	1	10	5	$D \neq 0$ **and** $D\&10 = 0$; continue to line 10.
$\cdots$ATGATC<u>G</u>AA$\cdots$	13	01000	2	10	5	$D \neq 0$ **and** $D\&100 = 0$; continue to line 10.
$\cdots$ATGATCG<u>A</u>A$\cdots$	13	01000	3	10	5	$D \neq 0$ **and** $D\&1000 \neq 0$; leave the loop and continue to lines 14, and 16–.
$\cdots$ATCG<u>A</u>A$\cdots$	17	01000	3	13	4	A suffix is found; *epos* and *last* are updated; the scanning direction changes.
$\cdots$ATCG<u>A</u>A$\cdots$	19	01000	4	13	4	$D\&10000 = 0$; not interesting, proceed with lines 22 and 23.
$\cdots$<u>A</u>TCGAA$\cdots$	23	10000	4	13	4	$D \neq 0$; proceed with lines 17, 18, and 9.
$\cdots$<u>A</u>TCGAA$\cdots$	19	10000	5	13	4	$D\&10000 \neq 0$; something interesting! A prefix or a match?
$\cdots$ATG<u>A</u>TCGAA$\cdots$	21	10000	5	13	4	$i = m(= 5)$; the else branch reports an occurrence at text position 9; continue to lines 24, 25, and 6–.
$\cdots$AATC<u>A</u>G$\cdots$	9	10001	0	17	5	The lowest bit is 1; continue from line 17 with BNDM.
$\cdots$AATC<u>A</u>G$\cdots$	19	10001	1	17	5	$D\&10000 \neq 0$; a prefix or a match?
$\cdots$AATC<u>A</u>G$\cdots$	20	10001	1	17	4	$i < m(= 5)$; it was a prefix, update *last*.
$\cdots$AAT<u>C</u>AG$\cdots$	23	00000	1	17	4	$D = 0$; continue to lines 24 and 25.
$\cdots$<u>C</u>AG$\cdots$	25	00000	1	21	4	$D = 0$; continue to line 6.

Implementation remarks. Note that on lines 10–13 the TNDM algorithm may address at most $m-1$ characters past t_n, the last character of text. This can be prevented by adding the following test in line 9:

9: **if** $(D\&1) = 0$ **and** *epos* $+ m - 1 \leq n$ **then**

The other possibility is to ignore spurious suffix and change line 14 in the following way:

14: **if** $D = 0^m$ **or** *epos* $+ i > n$ **then**

and allow references to $t_{n+1}, \ldots, t_{n+m-1}$. The third solution is to store in t_{n+1} a stopper character, e.g. null, which does not appear in any pattern.

If the interesting bits do not use the whole word i.e. $m < w$, then one has to be careful with tests like '$D \neq 0^k$'. As the result of a shift, some set bits may move beside the interesting area of bit-vector, and tests cannot be simplified to form '$D \neq 0$'. If the interesting bits are located on that edge in the shifting direction, the uninteresting bits fall off during the shift. Navarro and Raffinot use this concept successfully in their implementation of BNDM. In the pseudo-code of TNDM, all tests with 0^m can be simplified without extra masking.

Complexity. We consider only patterns that are at most w characters long. The preprocessing time is $\mathcal{O}(m + |\Sigma|)$. The worst case complexity of TNDM is clearly $\mathcal{O}(nm)$. The average case complexity of BNDM (and BDM) is $\mathcal{O}(n \log_{|\Sigma|} m/m)$. It is not difficult to see that the same is true for TNDM.

4.3 BNDM with q-grams – BNDMq

The BNDM algorithm (Algorithm 8) starts processing of every alignment by fetching the last text character in that alignment. If that character exists in the pattern, the BNDM algorithm falls to the slow loop. This happens quite often, when the pattern is long or the alphabet is small.

Technically, the state vector D gets the value of bit mask from table B corresponding to the last text character in the alignment. In the slow loop, the BNDM algorithm next fetches the penultimate text character (i.e. the last but one) in this alignment. The algorithm has to continue the loop, if the text character pair from the end of the alignment exists in the pattern.

Let us inspect further the behavior of the BNDM algorithm in process-

ing an alignment. If the last text character in the alignment appears also in the pattern, then another character from the text is fetched and used in the indexing of the table B. That bit-vector is then incorporated with the AND operation to the shifted state vector D, which is then tested. Let us assume that the two last text characters of an alignment are processed together. Then instead of two assignments to the state vector D and two tests, only one assignment and one test has to be made. As a consequence, the maximal shift length will decrease by one, because we cannot tell which one of the characters caused the state vector to become zero. However, this approach saves work on the average, because the probability of falling to the slow loop is typically smaller.

This idea extends naturally to more than just two characters. The use of q-grams is incorporated in Algorithm 14 called BNDMq. It was intro-

Algorithm 14 BNDMq($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $1 \leq q \leq m \leq w$

```
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 0
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j))
    /* Searching */
 4: i ← m − q + 1
 5: while i ≤ n − q + 1 do
 6:     D ← F(i, q)
 7:     if D ≠ 0 then
 8:         j ← i
 9:         first ← i − (m − q + 1)
10:         repeat
11:             j ← j − 1
12:             if D ≥ (1 << (m − 1)) then
13:                 if j > first then
14:                     i ← j
15:                 else
16:                     report occurrence at j + 1
17:             D ← (D << 1) & B[t_j]
18:         until D = 0
19:     i ← i + m − q + 1
```

duced in Publication [V]. $F(i, q)$ is a shorthand notation for instructions

$$B[t_i] \,\&\, (B[t_{i+1}] \ll 1) \,\&\, \cdots \,\&\, (B[t_{i+q-1}] \ll (q-1)).$$

Another difference to BNDM is a more simple instruction flow when the q-gram is not present in the pattern. This loop has been made as short as possible in order to quickly advance $m - q + 1$ positions in such a case.

Note that BNDMq does not have the $last$ variable storing the found prefix, but the variable i, which points to the text position corresponding to p_{m-q+1} in current alignment, is updated directly.

At the implementation level, the test starting the outer while loop can be removed by placing a copy of the pattern as a stopper beyond the end of the text [HuS91]. Then the end of the text is tested every time an occurrence of the pattern is encountered. When a match is encountered, it is unnecessary to process one more character from the text. We can break the loop and continue with shifting of the pattern.

4.4 More straight-forward versions of BNDM

4.4.1 Simplified BNDM: SBNDM

The inner repeat–until loop of BNDM checks one alignment of the pattern in the right-to-left order. At the same time, the loop recognizes prefixes of the pattern. The leftmost (and also the longest) one of the found prefixes determines the next alignment of the algorithm. When BNDM finds $t_h \cdots t_i$ which is a match or does not appear in P, there are two options for shifting. Let j be the smallest index such that $h < j \leq i$ holds and $t_j \cdots t_i$ is a prefix of P. Then the next alignment starts at t_j. If there is no such prefix, then the next alignment starts at $i + 1$.

In SBNDM, we shift as in BNDM in the case of a match. But if $t_h \cdots t_i$ does not appear in P, we skip examining of prefixes and set $h+1$ to be the start position of the next alignment. Naturally, this reduces the average length of shift, but on the other hand, the innermost loop of the algorithm becomes simpler. Our experiments show that SBNDM is most often faster than BNDM.

The pseudo-code of SBNDM is shown as Algorithm 15. Table B is initialized as in BNDM and TNDM. In the case of a match, the shift is s_0, which corresponds to the distance to the leftmost prefix of the pattern in

Algorithm 15 Simplified BNDM: **SBNDM**($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $m \leq w$

```
    /* Preprocessing */
 1: Initialize B and s_0
    /* Searching */
 2: pos ← 0
 3: while pos ≤ n − m do
 4:     j ← m;   D ← (~0) >> (w − m)                    /* D ← 1_2^m */
 5:     repeat
 6:         D ← (D << 1) & B[t_{pos+j}]
 7:         j ← j − 1
 8:     until D ≠ 0^m or j = 0
 9:     if D ≠ 0_2^m then
10:         report an occurrence at pos
11:         pos ← pos + s_0
12:     else pos ← pos + j + 1
```

itself. For example, s_0 is three for $P =$ abcab. In the case of a complete match, the shift is actually *restore*[1], computed with Algorithm 13. A more efficient computation of s_0 is shown as Algorithm 16. (Value of s_0 is actually same as the nett shift given by $\delta_2[1]$ in the Boyer–Moore algorithm.) If the proportional number of matches is not high, the algorithm runs equally fast with the conservative value $s_0 = 1$.

Algorithm 16 Computing s_0

```
    /* Preprocessing */
1: S ← B[p_m] ; s_0 ← m
2: for i ← m − 1 downto 1 do
3:     if S & (1 << (m − 1)) ≠ 0 then
4:         s_0 ← i
5:     S ← (S << 1) & B[p_i]
```

Note that it is possible to leave out from the SBNDM algorithm the test of j on line 8, because state vector D becomes always zero after m bitwise shifts. However this kind of version will need to examine one extra character after each match (immediately to the left of a match).

SBNDM was presented in Publication [II]. SBNDM is usually slightly faster than BNDM, especially for short patterns. Independently, Navarro [Nav01] has utilized a similar approach already earlier in the code of his

NR-grep. In the article [Nav01, pp. 1272–1273] there is also a description about how to apply similar simplified factor based shift in BDM algorithm, instead of shifts based on the longest prefixes.

4.4.2 SBNDMq

Next, we present SBNDMq, which is a revised version of SBNDM applying q-grams. The pseudocode, which has been developed from BNDMq, is shown as Algorithm 17.

The inner loops of BNDM and BNDMq contain two tests per fetched text character. The inner loop of SBNDMq has only one test. Because upon removing the tests on lines 12 and 13 in BNDMq (Algorithm 14), the loop runs in the case of a match one position further to the left. The loop does not go any further, because the $w - m$ leftmost bits of each $B[a]$ are zeros, where w is the word length and the m rightmost bits of D are zeros because of shifting left m times. Note that if there is an occurrence of the pattern in the beginning of the text, the algorithm reads the character t_0,

Algorithm 17 SBNDMq($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $1 \leq q \leq m \leq w$

```
    /* Preprocessing */
 1: for all c ∈ Σ do B[c] ← 0
 2: for j ← 1 to m do
 3:     B[p_j] ← B[p_j] | (1 << (m − j))
 4: Compute s_0 with Alg. 16
    /* Searching */
 5: i ← m − q + 1
 6: while i ≤ n − q + 1 do
 7:     D ← F(i, q)
 8:     if D ≠ 0 then
 9:         j ← i − (m − q + 1)
10:         repeat
11:             i ← i − 1
12:             D ← (D << 1) & B[t_j]
13:         until D = 0
14:         if j = i then
15:             report occurrence at j + 1
16:             i ← i + s_0
17:     i ← i + m − q + 1
```

which should be accessible or the beginning of the text should be processed otherwise. (Also BNDMq reads t_0 in such a situation. Nevertheless in the case of BNDMq it can be easily avoided at the implementation level.)

4.4.3 UFNDMq

Algorithms of BNDM and SBNDM and their descendents apply backward matching. The TNDM algorithm (section 4.2) uses backward and forward scanning. In this section, we introduce a new variation called FNDM (Forward Nondeterministic DAWG Matching) as Algorithm 18. A preliminary version of FNDM was introduced by Holub and Ďurian [HoĎ05]. The idea is to read every m^{th} character x of the text while x does not occur in the pattern. If x is present in the pattern, the corresponding alignments are checked by the naive algorithm. BNDM and its descendants apply the shift-and paradigm while FNDM uses shift-or.

Algorithm 18 FNDM($P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $m \leq w$

/* Preprocessing */

1: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow \sim 0$
2: **for** $j \leftarrow 1$ **to** m **do**
3: $\quad B[p_j] \leftarrow B[p_j]$ & $\sim(1 << (j-1))$

/* Searching */

4: $i \leftarrow m$
5: **while** $i \leq n$ **do**
6: $\quad D \leftarrow B[t_i]$
7: $\quad$ **while** $D \neq \sim 0$ **do**
8: $\quad\quad$ **if** $D < \sim 0 << (m-1)$ **then**
9: $\quad\quad\quad$ **if** $j = i$ **then**
10: $\quad\quad\quad\quad$ report occurrence at $i - m + 1$
11: $\quad\quad$ $i \leftarrow i + 1$
12: $\quad\quad$ $D \leftarrow (D << 1) \mid B[t_i]$
13: $\quad i \leftarrow i + m$

Next we extend FNDM to handle q-grams. Let $G(i, q)$ be a shorthand notation for instructions / expression

$$B[t_i] \mid (B[t_{i-1}] << 1) \mid \cdots \mid (B[t_{i-q+1}] << (q-1)).$$

If we replace the first occurrence of $B[t_i]$ on line 6 in Algorithm 18 by $G(i, q)$, we get FNDMq.

We will develop FNDMq further. The resulting algorithm is UFNDMq which is given as Algorithm 19. The letter U stands for upper bits because the algorithm utilizes those in the state vector D. Like FNDM, UFNDMq is a filtration algorithm. A candidate is checked by the naive algorithm only if at least q characters are correct. The reading step is q instead of m or 1 after a candidate has been processed. Checking can be done in any order.

Algorithm 19 UFNDM$q(P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n)$

Require: $1 \leq q \leq w - m + 1$

/* Preprocessing */

1: $mask \leftarrow (1 << (q-1)) - 1)$ /* $mask \leftarrow 0^{w-q+1}1_2^{q-1}$ */

2: **for all** $c \in \Sigma$ **do** $B[c] \leftarrow \sim(mask << m)$

3: **for** $j \leftarrow 1$ **to** m **do**

4: $B[p_j] \leftarrow B[p_j]$ & $\sim(1 << (j-1))$

/* Searching */

5: $t_{n+1}t_{n+2} \ldots t_{n+m} \leftarrow P$

6: $i \leftarrow 0; \quad D \leftarrow \sim 0$ /* $D \leftarrow 1_2^w$ */

7: **while** 1 **do**

8: **while** $(D|mask) = \sim 0$ **do**

9: $i \leftarrow i + m; \quad D \leftarrow (D << m) \mid G(i, q)$

10: $F \leftarrow (D \mid (1 << (m-1) - 1))$

11: **if** F **then**

12: Scan through unset (=0) upper bits in F

13: and check candidates starting at corresponding positions

14: **if** *end position* $> n$ **then**

15: Return

16: $i \leftarrow i + q; \quad D \leftarrow (D << q) \mid G(i, q)$

Checking is done if any of the higher than the $q-1$ lowest bits in D is not set. Those bits correspond to candidate positions.

Let us study an example. Let abcdefgh be the pattern, and let q be 4. Let us assume that the marked 4-grams have been read.

...xx$\overline{\text{xxab}}$cdef$\overline{\text{ghxx}}$xx...

Then the rightmost bits of D are computed as shown in Fig. 4.1. So the candidate abcdefgh should be checked.

Let us consider another example. Let q be 2. When bc of an occurrence of the same pattern has been read, i is advanced by 2 until the end of the pattern is recognized.

```
x:  ...00011111111
x:  ...100011111111
a:  ...1100011111110
b:  ...11100011111101
-------------------------
g:  ...1111111100010111111
h:  ...1111111110001111111
x:  ...1111111110001111111
x:  ...11111111110001111111
-------------------------
D:  ...11111111110111111111
```

Figure 4.1. Computation of D.

Notice that unlike the other q-gram algorithms UFNDMq works reasonably also on "undersized" patterns i.e. when $q > m$. Then it must be allowed to access characters before the beginning of text or better by evaluating the first value of D separately. A disadvantage of UFNDMq is that the pattern length is limited by $q + m \leq w + 1$.

4.5 Simultaneous two byte read

We adopted a similar approach as in Section 3.5 to BNDMq and SBNDMq. We developed three versions for both. BNDM2b/SBNDM2b reads a 2-gram as a 16-bit halfword. The value of $B[t_i] \,\&\, (B[t_{i+1}] << 1)$ is stored in a precomputed table g for each halfword. In BNDM4b/SBNDM4b the corresponding value of 4-gram is computed as $g[x_1] \,\&\, (g[x_2] << 2)$ where x_1 and x_2 are the halfwords and g is the same table used in the 2-gram version. In BNDM6b/SBNDM6b the value of 6-gram is computed as $g[x_1] \,\&\, (g[x_2] << 2) \,\&\, (g[x_3] << 4)$. From SBNDM4b, we made a modified version SBNDM2+2b, where a 4-gram is tested in two parts. If the first 2-gram does not exit in the pattern, we can shift $m - 1$ positions instead of $m - 3$ with 4-gram. Actually the table g is mostly zero. In the pattern there are $m - 1$ character pairs, and thus there are at most $m - 1$ nonzero entries in g.

During the preprocessing phase, we take care of endianess. The indexing of the table g is different. On a little endian machine, the bit-vector is stored to $(t_{i+1} << 8) + t_i$ and on a big endian machine to $(t_i << 8) + t_{i+1}$.

Simultaneous 4 byte read (as in Section 3.6) would cause mostly cross border read and would thus slow down. These kinds of algorithms would also need tables with 2^{32} entries and would be too big for practice. How-

ever, the space can be reduced by a transformation (Publication [IV, p. 421]).

4.6 Complexity

Provided that $m \leq w$, the worst case time complexity of BNDM is $O(mn)$, but the average time complexity is sublinear. The space complexity of BNDM is $O(|\Sigma|)$. It is straightforward to show that BNDMq and SBNDMq inherit these complexities. Also, UFNDMq is sublinear on the average and $O(mn)$ in the worst case.

We did not develop linear versions of our algorithms, because they would likely be in practice slower on the average than the standard versions as it is the case with BNDM [NaR00].

5. String matching on special data

5.1 String matching in chunked texts

We study exact string matching in special texts, which consists of consecutive fixed-length chunks where each position of a chunk has a character distribution of its own. This kind of setting can also be interpreted so that a chunk represents a character of a larger alphabet. If texts and patterns are of this kind, it may ruin the efficiency of common algorithms. We examine anomalies related to the Boyer–Moore–Horspool and Sunday's QS algorithms in this setting. In addition, we present two new algorithms. This section is mostly based on Publication [III].

5.1.1 Introduction

In most exact string matching algorithms, it is assumed that the characters are statistically independent of each other. If the text and the pattern contain similar regularities, well-known string matching algorithms may loose their efficiency. Lecroq [Lec98] considered string matching in texts with strong regularities. His texts were dumps of memory structures: arrays of numbers. Thus, the texts consist of consecutive fixed-length chunks where each position of a chunk has a character distribution of its own. In this section, we will use the term *chunked string* to indicate strings of this kind. Though Lecroq's texts were chunked, he did not consider the effects caused by chunks. He was more interested in the effect of the alphabet size. In what follows, we will present refinements to his work. We denote here the size of the alphabet with c.

We discovered anomalies in Lecroq's test results. Namely, with the two data sets he used, the Boyer–Moore–Horspool algorithm [Hor80] was unexpectedly slower than Sunday's QS algorithm [Sun90] (Sections 2.3

and 2.4), although these algorithms should behave in a similar way. We analyzed the reasons in detail. We found out that this phenomenon is due to the characteristics of the test data. This observation helped us to construct examples where the Boyer–Moore–Horspool algorithm works in $\mathcal{O}(n/m)$ and QS in $\mathcal{O}(nm)$ and vice versa.

5.1.2 Lecroq's experiments

Lecroq tested how well-known exact string matching algorithms behave on dumps of memory structures. He studied the effects of the alphabet size. Besides the traditional byte oriented approach, he considered alphabets where several consecutive bytes represent a character. So a chunk can be regarded as one character of a larger alphabet. Lecroq's algorithms and data are available on the Web[1].

We shortly describe Lecroq's two data sets [Lec98] which we use in our tests. *Shorts* or short integers consist of consecutive chunks of two bytes. *Doubles* or double precision numbers consist of consecutive chunks of eight bytes. Each chunk in Lecroq's data sets has been stored such that the high-order byte of the number is at the lowest address and the low-order byte at the highest address (i.e. bytes of the numbers are stored in big-endian order). The number of distinct values, counts of the most common values (max. frequency), and counts of zero for each byte are shown in short and double chunks in Table 5.1, where r denotes the probability that two randomly chosen bytes (in the given position or overall) are the same. In the shorts, the first byte follows a discrete uniform distribution. In the doubles, the proportion of zero bytes is considerable. The overall $1/r$ is moderate, but the different frequencies of zeros among positions of chunks makes searching more challenging. Lecroq's patterns were consecutive chunks extracted randomly from the texts.

Lecroq's conclusion was that the byte-oriented approach in searching was more efficient than the approach with extended alphabet (excluding the two shortest patterns of doubles). So we consider only the byte-oriented approach here. Lecroq used in his experiments the following algorithms: Boyer–Moore [BoM77], Boyer–Moore–Horspool [Hor80], Sunday's Quick Search [Sun90], and Tuned Boyer–Moore [HuS91]. We denote these algorithms by BM, BMH, QS, and TBM, respectively. Pairwise matching in the Boyer–Moore algorithm works inherently in the reverse

[1]URL: `http://www-igm.univ-mlv.fr/~lecroq/esmms.tar.gz`

Table 5.1. Statistics of the bytes of chunks.

DOUBLES	1	2	3	4	5	6	7	8	Overall
symbols	5	215	256	256	256	4	1	1	256
max.frequency	100152	6371	863	1344	9667	124889	200000	200000	536705
zeros	3	4	798	1344	9667	124889	200000	200000	536705
$1/r$	2.00	48.07	255.71	254.40	113.98	2.29	1.00	1.00	8.11

SHORTS	1	2	Overall
symbols	256	44	256
max. frequency	1564	12500	14064
zeros	1559	12500	14059
$1/r$	248.25	32.00	86.01

order, but the other implementations use C's library routine *memcmp* for matching. Quick Search and Tuned Boyer–Moore use an additional *guard test* [HuS91, pp. 1224–1225] before pairwise comparison (so called Raita's trick [Rai92]). Only the Tuned Boyer–Moore algorithm uses an unrolled fast skip loop (see Section 2.2) with factor 3. All the other implementations apply a simple skip loop searching for the last character of a pattern. Following pairwise matching, it is tested whether the potential occurrence of the pattern is at a correct alignment (phase) and does not extend beyond the end of the text.

Lecroq's data sets are big-endian. If Lecroq's tests were repeated with data corresponding to the same numbers on a little-endian machine, the results would be similar to those for texts of natural language. The length of the text of shorts is 400 000 bytes, and the length of the text of doubles is 1 600 000 bytes.

5.1.3 Why QS was faster than BMH?

Chunked texts and patterns may ruin the efficiency of common algorithms. Moreover, two algorithms may be inefficient in different cases. We illustrate this with the algorithms BMH and QS. These algorithms are closely related, and in practice their performances are similar on data which is not chunked.

In Lecroq's experiments QS used on the average 37% less processor time than BMH on the doubles, but he did not give any explanation for this phenomenon. The inspection of shift lengths reveals that something odd has happened: the average shift is over 14 for QS and less than 8

for BMH. From the characteristics of these algorithms, we know that if successive bytes do not (statistically) depend on each other, the expected length of shift for QS should be at most one longer than that for BMH.

A similar behavior was observed in the case of the short integers. QS was considerably faster than BMH for long patterns. Again there is a difference in the average length of shift. We show that the behavior in both cases is due to the characteristics of data.

In addition to BMH, the TBM does not perform well with Lecroc's data. Because TBM follows closely the behavior pattern of BMH, we decided not to consider it in detail.

The speed of QS and BMH should be almost equal

Theoretically, the speed of QS and BMH should be similar for long patterns in the case of random data. Let t_i be the text character aligned with the last character of the pattern. The main difference between the algorithms is that the shift is based on t_i in BMH and t_{i+1} in QS. If we assume that characters are independent of each other, the expected shift length [Bae89] of BMH is

$$\frac{1-(1-r)^m}{r},$$

where r is as in previous subsection. Similarly, it is straightforward to show that the expected shift length of QS is

$$\frac{1-(1-r)^{m+1}}{r}$$

When the pattern becomes longer, the expected shift lengths of both algorithms approach $\frac{1}{r}$. This means that the performance of BMH should get closer and closer to the performance of QS when patterns get longer. Note that pairwise comparisons of these algorithms are equally laborious.

If the characters come from the uniform discrete distribution of c different characters, the average shift approaches c in the case of long patterns. The improvement on expected lengths of shift should be quite clear while the pattern length m grows but still $m < c$ holds.

Anomaly of shorts

Lecroq's test data of short integers is not uniformly distributed. The text and patterns consist of chunks of two bytes. The second byte has a skew distribution among 44 values. A hexadecimal dump of the beginning of the text is given in Table 5.2.

In order to explain the different behavior of QS and BMH, let us consider a simple example. Let $T = (\#a\#b)^{n/4}$ and $P = (\#a\#b)^{m/4}$ be the

Table 5.2. The beginning of shorts.

```
bf04 5be9 a100 2051 38c4 60a1 3e10 7599
f6a4 edf9 6640 fe81 fea4 71f1 cf90 1909
36c4 ba89 1000 8331 6504 f5c1 9d90 5af9
2f64 b199 ce40 1e61 1be4 dc11 d810 ab69
9084 c329 d100 4011 d344 14e1 af10 7a59
0a24 df39 4840 5841 3b24 9031 5290 37c9
4c44 f5c9 6400 d6f1 0384 3e01 f290 53b9
06e4 f6d9 5440 2c21 dc64 0e51 bf10 3e29
ea04 d269 4900 c7d1 75c4 f121 e810 6719
```

text and the pattern, where # represents any character, e.g. T = *xaybcadbsayb*, P = *cadbfaxb*. In this case, a typical shift of BMH is two or four, but the average shift of QS is longer for long patterns. However, for a pattern $P = (a\#b\#)^{m/4}$ aligned differently, a typical shift of QS will be two or four, whereas the average shift of BMH would be longer.

In order to verify that this is the cause of the difference in Lecroq's test results, we ran a test with two possible alignments of patterns. The pattern set of Alignment 1 is the original one (patterns start on even bytes). Another set of 100 patterns, which start on odd bytes, was randomly picked from the text. This set is denoted Alignment 2. The results are shown in Table 5.3. BMH is clearly faster than QS in the latter case.

Table 5.3. Running times per pattern in seconds for short integers.

	Alignment 1		Alignment 2	
m	BMH	QS	BMH	QS
4	0.286	0.251	0.240	0.243
6	0.200	0.199	0.174	0.193
8	0.197	0.170	0.148	0.164
10	0.151	0.149	0.127	0.140
12	0.144	0.133	0.115	0.126
14	0.130	0.122	0.105	0.118
16	0.131	0.117	0.098	0.111
18	0.109	0.110	0.088	0.101
20	0.107	0.100	0.079	0.091
40	0.078	0.062	0.043	0.055
80	0.065	0.038	0.024	0.039
160	0.056	0.025	0.014	0.032
320	0.054	0.016	0.010	0.031
640	0.053	0.011	0.008	0.030
1280	0.058	0.011	0.010	0.032

The anomaly caused by Lecroq's data is not exceptional. When ASCII characters are represented in the UTF-16 (16-bit Unicode Transformation Format) coding, a similar situation appears on little-endian machines: rightmost bytes are zeros.

Anomaly of doubles

The double data is far from uniformly distributed. The text and patterns consist of chunks of eight bytes. In a chunk, the two last bytes are zeros and the value of the first byte is either 65 or 193 in practice. Because the penultimate character (i.e. the last but one) of each pattern is also zero, the shift of BMH on zero is one. Because the frequency of 193 and 65 is high in the first byte, the shift of BMH on 65 or 193 is likely seven and the shift on the other alternatives is $7+s$, where s is a multiple of eight. These characteristics lead BMH to a repetitive behavior in shifting where BMH has approximately two alignments of the pattern for each alignment of QS.

Moreover, the location of the double zero in the chunks of the pattern is critical. Table 5.4 shows the results with differently aligned patters. Note that BMH is clearly faster in five cases out of eight.

Table 5.4. Running times per pattern in seconds in Sparc for different alignments of doubles (patterns of 320 bytes).

offset	BMH	QS
0	1.389	1.016
1	0.884	0.257
2	0.181	0.237
3	0.176	0.230
4	0.172	0.237
5	0.178	0.264
6	0.238	2.133
7	2.060	2.016

The difference between QS and BMH can be even larger. Let us assume that at each alignment, the BMH algorithm checks the characters under the pattern in the order $p_m, p_1, p_2, ..., p_{m-1}$ and QS in the order $p_1, p_2, ..., p_m$ until a mismatch is found. Moreover, QS reads an extra character p_{m+1} at each alignment for shifting. It is known that the worst case complexity of the both algorithms is $\mathcal{O}(nm)$, whereas the best case complexity is $\mathcal{O}(n/m)$. Let us consider two examples:

(1) $P = a^m, T = (a^{m-1}b)^{n/m}$

(2) $P = a^{m-4}ca^3, T = (ba^{m-2}cb)^{n/m}$.

It is straight-forward to show that in the case (1) BMH performs in $\mathcal{O}(n/m)$ and QS in $\mathcal{O}(nm)$, but in the case (2) QS performs in $\mathcal{O}(n/m)$ and BMH in $\mathcal{O}(nm)$. These examples were tuned for forward pairwise checking in BMH and QS. It is not difficult to modify the examples for backward checking.

5.1.4 New algorithms

Fork. For long patterns, it is advantageous to base shifting of pattern on several characters (q-grams) in order to get longer shifts. There are several variations of Boyer–Moore and Boyer–Moore–Horspool algorithms which use 2-grams [Bae89, BeR99, KiS94, ZhT87]. 3-grams are not practical if no transformation is used for reducing the shift table. According to our experiments, consecutive characters yield a longer shift on the average than 2-grams with a gap, when the distribution of all bytes is uniform. Lecroq's double data is an example where also 2-grams, which are not consecutive, produce good results. We developed a variation which applies 2-grams which are not consecutive. The left character of the 2-gram is under the last position of the pattern and the right character is at a fixed distance from it to the right in order to make shift longer. This test can be combined with a skip loop. In the case where the character under the end of the pattern yields a shift that does not reach the other character, that shift is the final shift. This algorithm, shown as Algorithm 20 is called **Fork**. The parameter h is the offset of the right character of the 2-gram. The value $h = 1$ corresponds to a 2-gram of consecutive characters.

Fork can be viewed an extension of the Zhu–Takaoka and Berry–Ravindran algorithms [ZhT87, BeR99], which apply 2-grams of consecutive characters. The worst case complexity of Fork is clearly $\mathcal{O}(mn)$. The application of 2-grams never decreases the average shift length, so the average case complexity is at most the same as for BMH.

Sync. If we know the format of the data beforehand, it is possible to speed up string matching by taking this into consideration. For example, let a text and a pattern consist of consecutive chunks of u bytes. Then the shifts are necessarily multiples of u if non-aligned matches are not accepted. This was the case in Lecroq's experiments.

Moreover, the shift can be based on any byte of a chunk. So if the bytes of a chunk have different distributions, it is advantageous to base the shift on a byte position with a distribution as uniform as possible over all possible symbols.

Algorithm 20 Fork($h, P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n$)

Require: $m > h > 0$

/* Preprocessing */

1: **for all** $c \in \Sigma$ **do** $tempd[c] \leftarrow m$

2: **for** $i \leftarrow 1$ **to** $m - 1$ **do** $tempd[p_i] \leftarrow m - i$

3: ***shift*** $\leftarrow tempd[p_m]$; $tempd[p_m] \leftarrow 0$

4: **for all** $c1 \in \Sigma$ **do**

5: **if** $tempd[c1] < h$ **then**

6: **for all** $c2 \in \Sigma$ **do** $d[c1, c2] \leftarrow tempd[c1]$

7: **else**

8: **for all** $c2 \in \Sigma$ **do** $d[c1, c2] \leftarrow m + h$

9: **for** $i \leftarrow 1$ **to** h **do** $d[c1, p_i] \leftarrow m + h - i$

10: **for** $i \leftarrow 1$ **to** $m - h$ **do**

11: **if** $tempd[p_i] \geq h$ **then** $d[p_i, p_{i+h}] \leftarrow m - i$

/* Searching */

12: $t_{n+1} \cdots t_{n+2*m} \leftarrow P + P$ /* Stopper */

13: $j \leftarrow m$

14: **while** $j \leq n$ **do**

15: **repeat** $k \leftarrow d[t_j, t_{j+h}]$; $j \leftarrow j + k$ **until** $k = 0$

16: **if** $j \leq n$ **then**

17: **if** $t_{j-m+1} \cdots t_{j-1} = p_1 \cdots p_{m-1}$ **and** j is a multiple of u **then**
Report match

18: $j \leftarrow j +$ ***shift***

Algorithm 21 called **Sync** uses this idea. Sync applies the occurrence heuristics and is thus related to the Boyer–Moore–Horspool algorithm [Hor80]. Using the position of the least probable character in the pattern as a test position is also applied in the *Least Cost algorithm* [Sun90], where the whole pairwise matching is performed in the increasing order of probability of the characters. The worst case complexity of Sync is clearly $\mathcal{O}(mn)$, and the average case complexity is at most the same as BMH's.

Let h be the offset of the examined byte from the end of the pattern. We assume that h is less than u. By making some changes, it would be possible to allow larger values for h, but this extension might be useful only in rare special cases. Actually, the while loop starting from line 6 requires one test more: we need to check that we do not report matches beyond the end of the text.

On doubles Sync_4 outperformed all the other tested algorithms for all

Algorithm 21 Sync$(h, P = p_1p_2 \cdots p_m, T = t_1t_2 \cdots t_n)$

Require: $u > h \geq 0$ /* Let u be the width of a chunk */

/* Let h be the distance of the examined byte from the end */

/* Preprocessing */

1: **for all** $c \in \Sigma$ **do** $d1[c] \leftarrow m$

2: **for** $i \leftarrow u - h$ **step** u **to** $m - h - 1$ **do** $d1[p_i] \leftarrow (m - h) - i$

/* Searching */

3: $s \leftarrow p_{m-h}$

4: $t_{n+1}..t_{n+m} \leftarrow s^m$ /* Stopper for inner while */

5: $j \leftarrow m$

6: **while** $j \leq n$ **do**

7: **while** $t_{j-h} \neq s$ **do** $j \leftarrow j + d1[t_{j-h}]$

8: **if** $t_{j-m+1}..t_j = P$ **then** Report match

9: $j \leftarrow j + d1[s]$

patterns lengths. Fork$_4$ was the second fastest on the patterns at least 320 bytes long. For the shortest patterns, BM was faster. In these tests BM is an implementation of Algorithm 2 (BM_fast).

On short integers, TBM is the fastest for short patterns. Sync$_3$ is the fastest for patterns longer than 13 characters. This takeover point depends on the computing platform. On the AMD Athlon processor, the takeover point was around 20.

5.2 Long patterns

5.2.1 LBNDM

Navarro and Raffinot [NaR00, p. 12] introduced also a method of searching for patterns longer than w. They partitioned the pattern in consecutive subpatterns. All the subpatterns have w characters except possibly the rightmost one which gets the remaining characters. The leftmost subpattern is searched with the standard BNDM algorithm. Only when the match of the leftmost subpattern is found, the rest of an alignment is examined. The maximum shift is w.

We introduced in Publication [II] another approach called LBNDM for long patterns. LBNDM is able to make shifts longer than w. The pattern is partitioned in $\lfloor \frac{m}{k} \rfloor$ consecutive parts, each consisting of $k = \lfloor \frac{m-1}{w} \rfloor + 1$

characters. The $m - k\lfloor \frac{m}{k} \rfloor$ remaining character positions are left to either end of the pattern (or to both ends). This division implies k subsequences of the pattern such that the ith sequence takes the ith character of each part. The idea is to search first the superimposed pattern of these sequences so that only every kth character is examined. This filtration phase is done with the standard BNDM algorithm. Each occurrence of the superimposed pattern is a potential match of the original pattern and thus must be verified.

Note that the shifts of the LBNDM are multiples of k. To get a real advantage of shifts longer than in the approach of Navarro and Raffinot, the pattern length should be at least about two times w. On the other hand, this implies $k \geq 3$, which on DNA data turns out to be quite high. In the case of a small alphabet, a feasible solution could be to use q-grams instead of single characters, see [KST03].

5.2.2 BXS, BQL, and QF

In Publication [VII] we introduced three novel bit-parallel algorithms for search for long patterns. All of these algorithms apply q-grams.

The first algorithm is BXS (BNDMq with eXtended Shift). The pattern is cut into consecutive pieces and a superimposed pattern is formed by laying these pieces above each other. Constant q defines how many text characters are processed at the beginning of checking of every alignment. BXS is practical with large alphabets.

BQL (BNDMq Long) uses overlapping q-grams so that we effectively search for a pattern of $m - q + 1$ overlapping q-grams. Similarly to BXS we cut the q-gram pattern into pieces and superimpose them. Constant s is the hashing parameter used for q-grams. So the algorithm uses a hash table with $2^{q \cdot s}$ entries. Even surprisingly high values of q work well.

The third algorithm, QF (Q-gram Filtering), is similar to the approximate string matching algorithm by Fredriksson and Navarro [FrN04], which is not a BNDM based algorithm. As preprocessing we store for each *phase* i, $0 \leq i < q$, which q-grams occur in the pattern in that phase, i.e. start at the position $i + j \cdot q$ for any j. During searching we read consecutive q-grams in a window and keep track of *active* phases, i.e. such phases that all read q-grams occur in that phase in the pattern.

BQL and QF were shown to be optimal on the average.

6. Algorithm testing

Testing the performances of algorithms may at first seem easy. When we are interested in time, basically we need to store the clock time in the beginning, run the algorithm, store the clock time in the end, and calculate the difference. However, the results are surely valid only for the given input, for the implementation, compiler and hardware used, for the same workload, and for several other things.

The comparison of algorithms should of course be performed according to a good measurement practice. In the case of exact string matching algorithms it could mean following things. Verifying that algorithms work properly: whether all the matches are found, and whether the search always stop properly at the end of the text. It is not rare that the match in the beginning or in the end of the text is not recognized. When the measuring is focused on performance, it is reasonable to use at least some level of optimization in the compilation. The measurement should not disturb the work of algorithms. We think that the printing of matches during time measurement is questionable. Printing produces also additional overhead, which is partly unsynchronized. More generally one should investigate all possibly disturbing things for the measurement, and if possible, rule them out. It is crucial to measure the essential part of the algorithms. Typically in the case of string algorithms, the reading of data belongs merely to an outer part of the test setting. Therefore we think that times spent to reading should be excluded from time measurements unless we are comparing whole programs. However, it is not clear if preprocessing should be included in the measurements. The work required in preprocessing is a proportion of the whole search task, and the proportion is depending on the text length. For example Horspool reported that 'the timings do not include the work of initializing tables' [Hor80, p. 506].

It is also important to try to check the validity of the measurement

method, and the accuracy of results. One should check, if the order of input data or algorithms matters. It is important to find out, how accurately repeatable the results are.

Generally it is good to realize when comparison is focused to 'apples and oranges': are algorithms doing the same thing, or is some algorithm utilizing extra information or doing something which could output additional information.

Use of averages easily hides important details! Averages may tempt to biased or even wrong conclusions. Especially arithmetic mean is dangerous [FlW86].

6.1 Principles of testing

The choice of an implementation language for algorithms usually limits available features: the number of different data types and the exactness of requirements given to them varies typically. The programming language Java is defined precisely, but it lacks the unsigned integer data type, which is the best suitable for bit-vectors. On the other hand, the Java virtual machine adds an additional layer on the top of hardware. The programming language C is flexible, but its standard states quite loose requirements for the precision of integers. With assembly languages it would be possible to produce the most efficient machine code, but then the portability to another kind of hardware would be lost. We respect the implementations of algorithms made by others, and therefore avoided doing unnecessary modifications or changes to them. So the selecting of an implementation language for empirical comparison has been in practice bound to the language the majority of others uses. There are plenty of implementations of textbook style in Java. In most of them the practical performance has not been among the central goals. Nevertheless, the programming language C seems currently to be the de facto language for implementing efficient exact string matching algorithms.

The C language standard from year 1999 introduced the header file `stdint.h`. (The header file `inttypes.h` includes it.) The fastest minimum-width integer types designate integer types that are usually fastest to operate with among all the types that have at least the specified width. However, footnote 216 in the standard states 'The designated type is not guaranteed to be fastest for all purposes, if the implementation has no clear grounds for choosing one type over another, it will simply pick some

integer type satisfying the signedness and width requirements'. For example, the choice of certain data type may cause implicit type conversions that may ruin the otherwise fast operation. In our tests, the fastest minimum-width integer types had no clear effect on the performance. So they are not used in our implementations.

The exact-width integer types are ideal for use as bit-vectors. The typedef name `uint`N`_t` designates an unsigned integer type with a width of N bits. These types are optional. However, if the implementation (of a C compiler) provides integer types with widths 8, 16, 32, and 64 bits, it shall define the corresponding typedef names. Therefore, the exact-width integer types should be available in almost all the C compilers conforming to the C99 standard.

Test environment

We ran all our string matching tests using the test harness of Hume and Sunday [HuS91]. Recently another test harness SMART, the String Matching Algorithms Research Tool[1] was introduced. It has been built by Faro and Lecroq, and it contains currently 85 implementations of string matching algorithms. In addition there is also corpus of test texts and a testing framework. With SMART it is easy to quite automatically produce speed comparison of several algorithms and several data sets. The test harness of Hume and Sunday allows detailed testing of an individual algorithm and some of its variations even without changing the source code.

The test harness of Hume and Sunday has the following features:

- The preprocessing and the actual searching in the text are in separate functions with fixed names: `prep` and `exec`, respectively. This separation allows the precise measurement of the preprocessing time. However in most algorithms the pattern has to be copied to some local buffer so that pattern can be used in the verification of an occurrence. This copying is always made. Therefore all the measured preprocessing times contain copying.

  All data that are transfered from `prep` to `exec` are stored in `struct pat`. This guarantees for all local data structures in all algorithms a reasonable and equal behavior of the cache memory. Thus the comparison of algorithms is objective also in relation to the access of data structures.

[1] URL: `http://www.dmi.unict.it/~faro/smart/`

Placement to the structure `pat` may slightly increase the access time for individual variables, when they are accessed with a fixed offset from the base address instead of a direct memory address or even a register. This was often compensated inside the `exec` function by using local variables that were initialized from the value stored in that structure. Thus the compiler can easily optimize local variables away, if that seems to be more efficient.

- Each implementation of any algorithm is in a separate file, which forms a translation unit. It neatly limits the scope of identifiers. Also, the compiler's optimization can be improved upon, because the variables and other data structures of any search algorithm are not used elsewhere.

  When testing several algorithms in the same compiled program, there is always a risk that they interfere with each other. With this approach especially the data structures initialized during preprocessing are located close to each other, and thus prevent unnecessary cache misses.

  Adding a new implementation of a string matching algorithm requires possibly only an update to the scripts, but no changes to the program code of the test harness.

- The character type can be explicitly defined as `signed` or `unsigned` by defining the preprocessor macro `CHARTYPE` at compile time; e.g. in gcc `-D CHARTYPE= 'unsigned char'`. This feature is useful if in some processor the access times of signed and unsigned character type differ considerably. On the other hand, a related speed penalty may occur when converting the character type to some larger type and the speeds of zero extension and signed extension are not the same. Taking that into account usually triggers other changes in the implementation.

  In the programs in the `stringsearch` package by Hume and Sunday [HuS91] the use of the signed character type does not always work properly, because indexing of arrays does not work with negative values. This could be avoided by casting the value of an index to the unsigned type.

  In practice the use of preprocessor macro `CHARTYPE` seemed to have very little effect on the speed of a search.

- The data type used in shift tables can be defined through the typedef `TABTYPE`. This allows an easy use of a faster datatype in a particular

computer by defining a preprocessor macro `TABTYPE` at compile time; e.g. in gcc `-D TABTYPE='long int'`. The fastest minimum-width integer types described above might have effect on the performance. In practice, this typedef seemed to have little impact, so this feature is not always used in our implementations. This is in agreement with Hume and Sunday [HuS91, page 1232] who reported it.

- All printing is done after the time measurements of test runs. Therefore there should be no possibility of CPU time usage for unsynchronized I/O, which could be charged during time measurements.

  The reading operations are made and the files are closed before the search (of the first pattern) starts.

- In addition to the time measurements, the test harness offers also a possibility to collect and report some basic statistics about algorithm behavior on given data. That feature is enabled by defining the preprocessor macro `STAT` during compilation (e.g. in gcc `-DSTAT`). Collecting is based on counters programmed beforehand.

  The following counters in `struct stats` are used in the collection: number of character pair comparisons (`cmp`), number of otherwise fetched text characters (`jump`), average shift[2] lengths of the pattern (`step[]`), number of fallings to a slow loop (`slow`), number of enterings to other interesting parts of algorithm (`extra`, `extra2`; the latter was not in the original version), and pattern length (`len`). Of course `len` can store only the most recent pattern length. The name `jump` of jump may sound misleading. It could serve at least the two following purposes: Firstly it could mean the number of text characters that are used for shifting especially in skip loop. Secondly it could record the number of text characters that are fetched without fetching the corresponding character in the pattern. Interpretation of counters differs among algorithms.

  To improve locality in `struct stats` and possible errors caused by indexing past the end of the array `step[]`, we have permuted the latter to be the last item.

  In the unrolled fast skip loop there occurs one or more zero length shifts, when a potential matching position is found. One can argue that if zero length shifts are counted, the reported average shift length is unnecessarily short.

[2] Some authors use term stride for shift.

Mhashi [Mha05] measured the number of comparisons, including character comparison, number comparison, and character access comparison. Literally taken, this equals the number of conditional branches. Even more interesting in processors having branch prediction would be counting mispredictions and correct predictions (of conditional branches).

The `STAT` feature has not been added to all implementations that have been got from other researchers.

- There exist already good quality implementations of search algorithms: Hume and Sunday compiled their material in `stringsearch` package. It contains implementations of 37 search algorithms. About 30 of them have practical value.

We adapted the buffer sizes to the data sizes in the test harness. In addition we have added the call to `sched_setaffinity` function as described below.

In the implementations of algorithms the occurrences of the pattern are only counted but not printed out. For the sake of fairness, each algorithm has been implemented so that the exact location of each occurrence could be printed if wanted.

About implementation

In our implementations, we have tried to use efficient programming techniques. The programming style is similar as in the `stringsearch` package [HuS91], and in GNU grep[3]. Special care is taken of avoiding unnecessary assignments. In the literature the algorithms are usually presented with minimal number variables. For clarity this is a good convention. On the other hand, there are often parts in the expressions that remain constant in the search loop. For efficiency we have assigned them to local variables. We feel that this is very useful in boolean valued expressions. From array indices many compilers remove constant offsets by moving the base address accordingly: e.g. `A[j-1]` is probably equally fast as `B[k]`. We have also followed the old tradition by putting the `register` storage class to the definition of often used variables. Although it does not nowadays affect compiling according to our knowledge.

We have avoided use of functions. Only the following standard functions are used: The pattern is copied in `prep` with the `memcpy`, and if

[3]Version 2.5.3 and many earlier versions.

needed, beyond the text in `exec`. Large memory areas are initialized with the `memset` function. The `times` function is used for collecting CPU time usage.

In the implementations the characters are accessed using both pointers and indexing of array. (In the programming language C the difference between them is merely in appearance.) We noticed no differences in performance.

The change of the process from one processor core to another empties cache memories with various degree. (Often L3 and L2 caches are shared with several cores.) This would slow down reads from memory and induce annoying variation to the timing of test runs. To avoid it we have used the Linux function `sched_setaffinity` to bind the process to only one processor or core. Our test computer C (see details in Appendix B) had two separate processors. On it the use of the function `sched_setaffinity` reduced substantially variation in time measurements.

Many modern processors are able to use the dynamic CPU frequency scaling in order to conserve power. Changing the CPU speed during a performance test changes also the relative speed of CPU and memory bus. This would favor less CPU intensive algorithm implementations, but more importantly it would add variance to time measurement. Therefore the CPUFreq drivers (of Linux) were disabled. HyperThreading is available in the test computer D. We ran several tests with and without HyperThreading, but we did not notice any difference. (We suspect that the scheduling policy was smart enough.) During test runs the computers were connected to the network, and the cron jobs were running, but there was no other workload. On test computer A even windowing system Gnome was not running.

All bit-vectors are coded to use the type `Bitv`, which is defined to be either `uint32_t` or `uint64_t`. The aim is to use the largest unsigned integer type that can be operated with single machine instructions. The number of bits in bit-vectors can be overridden by defining a preprocessor macro `BITS` at compile time; e.g. in gcc `-D BITS=32` or `-D BITS=64`. The following piece of code is used in defining the type `Bitv`

```
#ifndef BITS
# if(4294967295UL==ULONG_MAX)
#  define BITS  32
# else
#  if(18446744073709551615UL==ULONG_MAX)
```

```
#   define BITS 64
#  else
#   error 'this implementation has not been tested for this number of bits'
#  endif
# endif
#endif

#if(BITS==64)
typedef uint64_t Bitv;
#elif(BITS==32)
typedef uint32_t Bitv;
#else
typedef uint32_t Bitv;
# error 'strange value defined for BITS'
#endif
```

Considerable effort has been put in getting good implementations of exact string matching algorithms in the C language from their designers, or publicly available resources, and sometimes even copied from the original articles. Many implementations of algorithms seem to be made for clarifying their operations, and thus are not following a normal efficient programming technique. The implementations tested in this thesis follow the same limitations of the problem. For example they use no additional information; such as a character not occurring in the text. From some algorithms competing versions were made to verify their efficiency. If different versions had clearly different superiority areas, several versions were taken to the final test runs.

Necessary modifications were made to the program codes to fit them to the test harness:

- Replacing the nonstandard features (e.g. C++ I/O);

- Separating the preprocessing (`prep`) from the searching in the text (`exec`); changing the names of functions to those

- Adding fundamental tests for the input parameters (e.g. checking that the pattern is not shorter or longer than it is possible to handle with the implementation)

- Removing of use of global variables by defining all variables local to the module (with `static`) or defining them inside the functions.

- Usually all the variables defined in a module were grouped inside structure `pat`. This modification may slightly increase the access time for individual variables as discussed on page 76.

Measuring processor time

The C standard library offers only the `clock` function for watching the CPU time usage of processes. To determine the time in seconds, the value returned by the `clock` function should be divided by the value of the macro `CLOCKS_PER_SEC`. Additionally POSIX[4] declares that `CLOCKS_PER_SEC` is defined to be one million in `<time.h>`, and also that 'the resolution on any particular system need not be to the microsecond accuracy'. On the Linux platform the typical resolution is 10 milliseconds. Lecroq [Lec07] used the `clock` function in his tests.

POSIX offers the `times` function for getting process (and waited-for child process) times. The number of clock ticks per second can be obtained using: `sysconf(_SC_CLK_TCK)`[5]. The test results by Hume and Sunday [HuS91] were given as user time (`tms_utime`) fetched with the `times` function. So the system time was excluded. (Occasionally we have checked that there is not any hidden use of the system time, but have never noticed.) The `times` function was used also in the tests results presented here.

The POSIX function `clock_gettime` returns the current value *tp* for the specified clock, *clock_id*. The struct *tp* is given as a parameter. If symbol `_POSIX_CPUTIME` is defined, implementations shall support the special *clock_id* value `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process. The resolution of the given clock at the function `clock_gettime` is provided with the `clock_getres` function.

The POSIX function `gettimeofday` returns the current time, expressed as seconds and microseconds since the Epoch. Applications should use the `clock_gettime()` function instead of the obsolescent `gettimeofday()` function.

[4] IEEE Std 1003.1-2008

[5] Hume and Sunday used the symbol `CLK_TCK` (defined in `<time.h>`). But in POSIX.1-1996 it is mentioned as obsolescent.

Precision of individual search

When the digital clock moves evenly, it is safe to assume that its value is incremented on regular fixed intervals. These time intervals are called *clock ticks*. In computers they are typically so long that during individual tick several instructions are executed. It this section term clock tick refers to the precision of time measurements. So it is assumed that the processor time increases one tick[6] at a time. When the measuring of a time interval starts, we fetch the last updated value of the clock, but a part of the current clock tick may be already spent. This time follows the continuous uniform distribution $[0, 1]$. So its mean is 0.5 and variance $1/12$. Respectively when the measuring of a time interval ends, possibly a part of the current clock tick may be unspent. This slice follows the continuous uniform distribution $[-1, 0]$.

Thus the time measurement with clock ticks has an inaccuracy which is the sum of two error terms following the above mentioned distributions. When the length of the measured interval is at least one clock tick, the probability density function of the sum is

$$f(x) = \begin{cases} 1 + x & \text{if } -1 \leq x \leq 0, \\ 1 - x & \text{if } 0 \leq x \leq 1, \\ 0 & \text{otherwise.} \end{cases}$$

The mean of the inaccuracy caused by clock ticks is 0 and the variance $1/6$.

The variance of the inaccuracy caused by clock ticks becomes relatively smaller, when the count of clock ticks increases. An easy way to achieve this is to use a longer text, which is at the same time more representative (statistically). However, it is not advisable to use concatenated mul-

[6] In Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2, Chapter 17.13 it is told (URL: `http://download.intel.com/products/processor/manual/253669.pdf`: The X86 or X86_64 architectures have the RDTSC instruction for reading the time-stamp counter. Processor models increment the time-stamp counter differently:

- On older models the time-stamp counter increments with every internal processor clock cycle.
- On recent processor models the time-stamp counter increments at a constant rate.

Constant time-stamp counter behavior ensures that duration of each clock tick is uniform.
However, this doesn't guarantee that the increment is one. Therefore the RDTSC instruction may return results that are multiples of 'clock tick's discussed here.

tiples of a short text, because it is probable that the shifts of patterns start to follow similar sequences in such a case. This happens surely, if the pattern matches the text, assuming that the pattern is moved from left to right, and the shifting logic does not have any random behavior. (Nevertheless this does not happen on algorithms using a constant shift $q > 1$: e.g. Average-Optimal-Shift-Or and Fast-Average-Optimal-Shift-Or algorithms [FrG05].) Therefore the text produced with concatenation will with a high probability show the same statistical peculiarities as the original text element.

There are also other causes for inaccuracy. Generally a context switch from a process to another produces more or less delay. On a single CPU system it is very hard to minimize them. We have noticed that on modern multicore processors it is possible to get more accurate measurement of used CPU time than with singlecore processors spending similar number of clock ticks. The variance caused by other processes becomes relatively smaller, when the measured time intervals get longer. Then the results are more accurate.

Precision of search with a pattern set

The search with a pattern set brings still one source of variance to the time measurements. Search for some patterns is more laborious. On the other hand some algorithms work more efficiently on a certain type of patterns. This joint impact of the patterns and the algorithms can be seen as samples of the all possible cases between the worst and the best cases of a given algorithm. This kind of variance is minimal with the Shift-Or algorithm. In practice with it only a large number of occurrences causes small variation. One may also argue that if certain algorithms have a similar search time, the algorithm with smallest variance can be regarded the best.

When several successive time measurements are done within a relatively short period, it is possible that the unused time slice before the next clock tick is utilized in the next time measurement. Thus the time measurements may not be completely independent. In our test setting preprocessing and search are alternating. If the measured time intervals are at least a few clock ticks, there is always so much variance, that it is unlike that these surplus times cumulate more to either preprocessing or search.

Let us consider the variance of the mean. If the measurements are

(statistically) independent, then the variance of the sum of times is the sum of the variances of individual time measurement. If the measurements have the same variance, and if they are independent of each other, the variance V of the mean of r measurements is

$$V\left(\frac{X_1 + X_2 + \ldots + X_r}{r}\right) = \frac{1}{r^2}V(X_1 + X_2 + \ldots + X_r) = \frac{r \cdot V(X_1)}{r^2} = \frac{V(X_1)}{r}$$

If the measured times contain only a few clock ticks, this could cause severe bias to the measurements: the results are more likely too large than too small. Fortunately the preprocessing times have small variance (especially on patterns of equal length), and they are typically much shorter than search times[7].

6.2 Experimental comparisons

Next we report about experimental tests of the exact string matching algorithms. Our intention is not to make an extensive comparison of the exact pattern matching algorithms, but to demonstrate the practical performance of the algorithms presented here in a fair test setting. Therefore we have included several algorithms serving as reference methods.

Artificially generated test data may reveal asymptotic characterization of algorithms. Especially the commonly used discrete uniform distribution of characters is quite rare in practice. The exceptions are nucleotide sequences and compressed data. However, we are not aware of any model which would in a realistic manner represent some common type of real data. Another open question is, what numerical values should be used for representing the characters of the generated alphabet. This has great impact to the algorithms using hashing. Therefore no artificially generated text data is used in these tests.

Natural language is perhaps most commonly associated with the term text. Therefore English text is the most essential test data here. It would be interesting to make tests on a text of some other language with a larger alphabet. If the characters were coded in UTF-8, and roughly about a quarter of the codes were longer than one byte, then the byte distribution would be challenging. For a given language there is often one leading byte (in the UTF-8 encoding) that is most common. So there would be

[7]The test harness reports with the flag -% the shortest search time and 20 other regular interval samples from the sorted search time list. So it is easy to notice too small tick counts, which would produce unnecessary error variance.

another common character in addition to the space. Unfortunately we did not manage to find such a text of at least one megabyte long.

DNA sequences are typically long. So the time spent for string searching is remarkable. Therefore DNA sequences are a beneficial application area for the exact pattern matching algorithms. The small alphabet and a nearly uniform character distribution offer a unique platform to study the variation in search times more closely.

In Publication [VII] we present experimental results on long patterns. Some of those algorithms would be competitive with long DNA patterns, but they were not included in the tests of this chapter.

64-bit CPUs have existed in workstations and servers since the early 1990s. Currently numerous computers are working on the x86-64 instruction set which is a 64-bit extension of the 32-bit x86 instruction set. These processors have also a so-called 32-bit mode, which is capable to run programs of the original architecture. The doubled integer width in x86-64 allows efficient operation for the bit-parallel algorithms also on longer patterns. The larger number of addressable registers speeds up most programs. To our astonishment in preparation of Publications [V], [VI], and [VII] it became clear, that there are surprising speed differences in the programs that were compiled to run in the 32-bit mode compared to the same programs compiled to run in the 64-bit mode. Therefore we felt necessary to run current tests in both the 32-bit mode and the 64-bit mode. It should be noted that in the 32-bit mode 32-bit bit-vectors are used, and in the 64-bit mode 64 bit bit-vectors are used on all pattern lengths. So the memory area, that is initialized in the 64-bit mode, is almost double.

For the sake of completeness, we have included the technical details of the computers used in test runs in Appendix B. The naming and short descriptions of the implementations of algorithms used in tests are listed in Appendix A.

Comparison on DNA data

The tests with DNA data are divided into two parts. The tests results with the short patterns are collected in Appendix C. The test results on DNA with pattern lengths up to 60 nucleotides are collected in Appendix D. Algorithms TNDM and SVM were left out of the tests because the newer algorithms are clearly faster.

With short DNA patterns it is possible to get quite accurate estimates about the variance of search time. The central idea is to use all the pos-

sible combinations of a, c, g, and t as patterns. Other symbols are extremely rare in real data. (In RNA u replaces t.) The distribution of the four most common nucleotides is quite uniform, but the distribution of oligonucleotides is not as uniform. On the other hand, one may want to search for the rarest oligonucleotides of given length. An interesting side effect in this approach is that if the text contains only symbols a, c, g, and t, then the total number of matches for the whole pattern set is $n - m + 1$. On a DNA text, we tested the pattern sets separately for $m = 2, 3, 4, 5, 6$. (For larger values of m the number of patterns grows unpractically large.)

The fastest algorithms in the 32-bit mode with short DNA patterns on test computer B:

$m = 2$: FAOSO(1,24), EBOMb, SBNDMq2b, Shift-Or
$m = 3$: Fast-Shift-Or, FAOSO(1,24), Shift-Or, FAOSO(1,16), EBOMb
$m = 4$: Fast-Shift-Or, FAOSO(1,24), FAOSO(1,16), Shift-Or, SBNDMq2
$m = 5$: Fast-Shift-Or, SBNDMq4b, FAOSO(1,24), BMH4b
$m = 6$: SBNDMq4b, Fast-Shift-Or, BMH4b, FAOSO(2,14)

The fastest algorithms in the 64-bit mode with short DNA patterns on test computer B:

$m = 2$: Shift-Or, SBNDMq2b, EBOMb, EBOM, Fast-Shift-Or
$m = 3$: Fast-Shift-Or, UFNDM3, Shift-Or, UFNDM4, EBOMb
$m = 4$: Fast-Shift-Or, UFNDM4, UFNDM3, UFNDM5, SBNDMq2+2b
$m = 5$: Fast-Shift-Or, UFNDM4, SBNDMq4b, UFNDM5, SBNDMq3
$m = 6$: SBNDMq4b, UFNDM4, UFNDM5, Fast-Shift-Or, SBNDMq3

Fast-Shift-Or is the fastest algorithm with short DNA patterns of length 3–5. The implementation uses unrolling factor 8, which seems to be too large on dinucleotides. On test computer D results are practically the same, but Shift-Or is the fastest also in the 32-bit mode on dinucleotides. (Times are not shown here.) Test computer A can execute only 32-bit code. There SBNDMq2b is the fastest algorithm with DNA patterns of length 2–4. On pattern length 5 BMH3b and BMH are the fastest, and on pattern length 6 SBNDMq4b is still faster than them. (Times are not shown here.)

With DNA patterns longer than 32 nucleotides in the 32-bit mode the algorithm KS is the fastest. Lecroq's hashing scheme follows quite close behind, but old BMH4 and newer BMH4b are equally fast as HASH4 and

HASH5. Up to 32 nucleotides long patterns the bit-parallel algorithms are the best: on pattern lengths between 20–30, SBNDMq6b is the fastest. From previous test we know that Fast-Shift-Or is the fastest up to 5 nucleotides, but after that SBNDMq4b takes the lead.

Notice the huge speed up of FAOSO with a sample distance larger than 3 (i.e. the first parameter), when patterns get longer. Clearly FAOSO would need larger than 32 bits long bit-vectors. Unfortunately we do not have a 64-bit version of FAOSO.

Results for UFNDM4, UFNDM5, and UFNDM6 are quite good on nucleotide patterns of length 10 and 20 on test computer D. (Times are not shown.) On test computer A BMH4 and BMH4b are the two fastest, while HASH4 and KS are following them on pattern lengths longer than 32.

In the 64-bit mode the bit-parallel algorithms are the best. On 5 nucleotides long patterns Fast-Shift-Or is best, but UFNDM4 is quite close to its performance. (Here in the pattern set containing 5 nucleotides long patterns there are 200 patterns, of which 176 are unique.) Starting from 20 nucleotides long patterns SBNDMq6b is the fastest. On the area between SBNDMq4b is the fastest. The modified Berry–Ravindran algorithm was always faster than the original. Results are similar also on test computer D; (times are not shown here.) Interestingly BMH4 is faster than BMH4b with patterns longer than 30 on test computer B, but not on test computer D.

In Table D.2 (in Appendix D) are shown also the sample variances. The coefficient of variation is defined as the ratio of the standard deviation to the mean. The coefficient of variation is considered an objective indicator of variation. Therefore one should note that the search times are largest with the shortest patterns. Still there is a lot of variation with so uniform distribution that DNA sequences have.

Comparison on English data

The space character separating words in texts in many natural languages adds an essential spice to the exact string matching. The space character is very frequent and its occurrences have an own kind of distribution. In Publication [VI] (in Section 4.1) we reported that the performance of algorithms changes when patterns of same length are any kind of substrings instead of words. Even the relative performance order of algorithms sometimes changes. One may argue that patterns used in search in natural language texts are typically words, and therefore they are statistically

more representative. Nevertheless, it is also quite common to search two (or even more) consecutive words. We feel that generally randomly picked substrings make a biased pattern set for searches from natural language texts.

The test with English text is reported in Appendix E. The pattern sets consists of full words. In the 64-bit mode EBOMb was the fastest one, and in the 32-bit mode among the three best implementations on pattern lengths up to 12. SBNDM2+2b was among the three best on pattern lengths longer than 3 characters and fastest or second fastest on pattern lengths longer than 10. SBNDM2b was mostly even more competitive; up to 10 characters it was always faster than SBNDM2+2b. UFNDMq is much faster in the 64-bit mode than in the 32-bit mode. Also SBDNMq2 is better in the 64-bit mode. FSBb+ and SBNDMq2uf are extremely similar algorithms. With the shortest patterns the use of the lookahead character seems beneficial. Lecroq's hashing scheme can be successfully applied to 2-grams: HAHS2c and HAS2i are clearly faster than HASH3.

In the modified Berry–Ravindran algorithm, BRX, the shifts are based on the last character of the current text alignment t_s and the next character t_{s+1}, instead of t_{s+1} and t_{s+2} as in the original Berry–Ravindran. To our surprise the BRX is slower than BR on English text. We are not aware of any report about so strong dependence of adjacent characters, that it would explain this phenomenon. Moreover it is possible that blank characters that do not exist in the patterns may also affect this way. It is clearly questionable that the discrete uniform distribution would be a realistic model for English texts and generally for natural language texts.

On test computer D the SBNDM2+2b was fastest or second fastest on all pattern lengths longer than 3 characters both in the 32-bit and the 64-bit mode. EBOMb was among the three best on all pattern lengths. (Times are not shown.)

Test computer A has old 32-bit hardware (so the preprocessing was repeated 2000 times and searches 10 times). On it HASH2i was the fastest on pattern lengths 9–13. The versions on FSB showed good performance on longer patterns than 5 characters. (Times are not shown.)

7. Discussion and conclusions

Several new algorithms for exact string matching have been presented in this thesis. Among the created algorithms are novel ones and variations of previous algorithms. Our extensive practical experiments show that many of our algorithms are faster than earlier algorithms in several cases. In our tests we used English, DNA, and (in the publications) random binary text with a wide range of pattern lengths. In addition, we discuss the principles of fair testing of string matching algorithms.

Many of our algorithms start treatment of every alignment by processing a q-gram. This approach saves plenty of tests and accelerates the moving to the next alignment. This shift may be shorter than otherwise, but on the average this speeds up processing. The q-gram technique can also be regarded as an example of the power of branchless computation [Knu11, pp. 180-181].

Based on the results of speed tests on various kinds of data, we noticed that the following rule of thumb holds roughly in the case of many algorithms: start processing an alignment with at least two characters when $m \geq \sqrt{e}$, where $1/e$ is the probability that two random characters are the same (i.e. e is the size of the effective alphabet). The EBOM algorithm is an exception. It works excellently even with short patterns.

For different kind of text types and for different pattern lengths the best algorithms are also different. The same applies also to some extent for different kind of computer hardware. It is unlike that there would ever be a single best exact string matching algorithm except perhaps some specialized machine instruction (discussed in subsection 2.2).

The performance of processors has lately increased, because of development in branch prediction and speculative execution of processors. Therefore the pipeline of processors should not stop so often as before. Nevertheless, it has been shown here that reducing the number of conditional

branches improves the search speed.

The Boyer–Moore–Horspool algorithm has perhaps unnecessarily low esteem. Its performance clearly suffers, when the pattern contains common characters. Our BMHq modification smoothens the performance and makes it competitive especially on DNA texts.

Shift-vector matching, SVM, is a novel approach to bit-parallel string matching. It examines quite few text characters, but shifting of the pattern is laborious. The SVM algorithm has potential in search in repetitive texts.

The BNDM algorithm utilizes bit-parallelism, which works very well on modern processors. Various simplifications of it have been shown to be faster in practice than the original algorithm. The use of q-grams can be successfully incorporated to both BNDM and SBNDM variations. At least with these algorithms simplification and use of q-grams interact well.

When the pattern length is close to register width w, the FSB algorithm performs well especially on natural language. The SBNDMq algorithm family is competitive in pattern lengths up to the register width w. Use of simultaneous 2-byte read speeds up all of these algorithms further [PeT11].

In Publication [VII] there was a comparison on long patterns that are at least 25 characters long. The QF algorithm was the fastest except on the random data with a large alphabet. BXS was in that case sometimes a little slower and in other times a little faster than QF. It should be noted that EBOM was once the fastest and always competitive with a large alphabet.

Two essential guidelines of exact string matching became clear also during this work. It is important to keep the skip loop running. And secondly the number of fetched text characters matters very little to the practical speed. “Clearly, the run time metric is unrelated to the character comparison metric.” [HuS91, p. 1242] It is also worthwhile to notice the difference between fetching a byte (as in bit-parallel algorithms) and a comparison of characters from the pattern and the text.

References

[ACR99] C. Allauzen, M. Crochemore, and M. Raffinot: Factor oracle: a new structure for pattern matching. In J. Pavelka, G. Tel, and M. Bartošek, editors, *26th Conference on Current Trends in Theory and Practice of Informatics, SOFSEM'99* , volume 1725 of *Lecture Notes in Computer Science*, pages 295–310, Springer-Verlag, Berlin, 1999. doi:10.1007/3-540-47849-3_18

[AlD86] L. Allison and T.I. Dix: A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, **23**(6):305–310, 1986. doi:10.1016/0020-0190(86)90091-8

[ApG86] A. Apostolico and R. Giancarlo: The Boyer–Moore–Galil string searching strategies revisited. *SIAM Journal on Computing* **15**(1):98–105, 1986. doi:10.1137/0215007

[Arndt] J. Arndt: jj's useful and ugly BIT WIZARDRY page. *URL:* `http://www.jjj.de/bitwizardry/bitwizardrypage.html`

[Bae89] R. Baeza-Yates: Improved string searching. *Software: Practice and Experience*, **19**(3):257–271, 1989. doi:10.1002/spe.4380190305

[Bae89w] R. Baeza-Yates: String Searching Algorithms Revisited. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Proceedings of the Workshop on Algorithms and Data Structures, WADS '89, Lecture Notes in Computer Science*, **382**:75–96, Springer-Verlag, Berlin, 1989. doi:10.1007/ 3-540-51542-9_9

[BaG92] R. Baeza-Yates and G.H. Gonnet: A new approach to text searching. *Communications of the ACM* **35**(10):74–82, 1992. doi:10.1145/135239.135243

[BCW90] T.C. Bell, J.G. Cleary, and J.A. Witten: *Text Compression*. Prentice Hall, 1990. ISBN-10: 0-13-911991-4.

[BeR99] T. Berry and S. Ravindran: A fast string matching algorithm and experimental results. In J. Holub and M. Šimánek, editors, Collaborative Report DC-99-05 *Proceedings of the Prague Stringology Club Workshop '99*, Czech Technical University, Prague, Czech Republic, Collaborative Report DC-99-05, pp. 16–28, 1999. *URL:* `http://www.stringology.org/event/1999/psc99p2_ps.gz`

[BoM77] R.S. Boyer and J S. Moore: A fast string searching algorithm. *Communications of the ACM*, **20**(10):762–772, 1977. doi:10.1145/359842.359859

[CaF03] D. Cantone and S. Faro: Fast-Search: A New Efficient Variant of the Boyer–Moore String Matching Algorithm. In K. Jansen, M. Margraf, M. Mastrolilli, and J.D.P. Rolim, editors, *Proceedings of the Second International Workshop on Experimental and Efficient Algorithms, Second International Workshop, WEA 2003, Lecture Notes in Computer Science*, **2647**:47–58, Springer-Verlag, Berlin, 2003. doi:10.1007/3-540-44867-5_4

[CaF05] D. Cantone and S. Faro: Fast-Search algorithms: New efficient variants of the Boyer–Moore pattern-matching algorithm. *Journal of Automata, Languages and Combinatorics*, **10**(5/6):589–608, 2005.

[ChLe04] C. Charras and T. Lecroq: *Handbook of Exact String Matching Algorithms*. College Publications, 2004. ISBN-10: 0-9543006-4-5. ISBN-13: 978-0-9543006-4-7.

[Col91] R. Cole: Tight bounds on the complexity of the Boyer–Moore string matching algorithm. In *Proceedings of the Second Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'91) (San Francisco, California, United States, January 28–30, 1991)*. Society for Industrial and Applied Mathematics, Philadelphia, PA: 224–233, 1991.

[CrR94] M. Crochemore and W. Rytter: *Text Algorithms*. Oxford University Press, 1994. ISBN-10: 0-19-508609-0.

[Döm64] B. Dömölki: An algorithm for syntactic analysis. *Computational Linguistics*, 3:29–46, 1964. Hungarian Academy of Sciences, Budapest.

[FaL08] S. Faro and T. Lecroq: Efficient variants of the backward-oracle-matching algorithm, In J. Holub and J. Žďárek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 146–160. Czech Technical University in Prague, Czech Republic. *URL:* http://www.stringology.org/event/2008/p14.html

[FaL09] S. Faro and T. Lecroq: Efficient variants of the backward-oracle-matching algorithm, *Int. J. Found. Comput. Sci.*, **20**(6):967–984, 2009. doi:10.1142/S0129054109006991

[FaL10] S. Faro and T. Lecroq: The exact string matching problem: a comprehensive experimental evaluation. *CoRR abs/1012.2547*, 2010. *URL:* http://arxiv.org/pdf/1012.2547v1

[FJS07] F. Franek, C.G. Jennings, and W.F. Smyth: A simple fast hybrid pattern-matching algorithm. *Journal of Discrete Algorithms*, **5**(4):682–695, 2007. doi:10.1016/j.jda.2006.11.004

[FlW86] P.J. Fleming, J.J. Wallace: How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, **29**(3):218–221, 1986. doi:10.1145/5666.5673

[Fre03] K. Fredriksson: Shift-or string matching with super-alphabets. *Information Processing Letters*, **87**(4):201–204, 2003. doi:10.1016/S0020-0190(03)00296-5

[FrN04] K. Fredriksson and G. Navarro: Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, **9**(1.4):1–47, 2004. doi:10.1145/ 1005813.1041513

[FrG05] K. Fredriksson and Sz. Grabowski: Practical and optimal string matching. In M. Consens and G. Navarro, editors, *String Processing and Information Retrieval, 12th International Conference, SPIRE 2005, Lecture Notes in Computer Science*, **3772**:376–387, Springer-Verlag, Berlin, 2005. doi:10.1007/11575832_42

[FrG09] K. Fredriksson and Sz. Grabowski: Average-optimal string matching. *Journal of Discrete Algorithms*, **7**(4):579–594, 2009. doi:10.1016/j.jda.2008.09.001

[Gus97] D. Gusfield: *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN-10: 0-521-58519-8. ISBN-13: 978-0-521-58519-4.

[HoĎ05] J. Holub and B. Ďurian: Fast variants of bit parallel approach to suffix automata. *URL:* http://www.cri.haifa.ac.il/events/2005/string/presentations/Holub.pdf, 2005. Talk given in The Second Haifa Annual International Stringology Research Workshop of the Israeli Science Foundation.

[Hor80] R.N. Horspool: Practical fast searching in strings. *Software: Practice and Experience*, **10**(6):501–506, 1980. doi:10.1002/spe.4380100608

[HuS91] A. Hume and D. Sunday: Fast string searching. *Software: Practice and Experience*, **21**(11):1221–1248, 1991. doi:10.1002/spe.4380211105

[KiS94] J.Y. Kim and J. Shawe-Taylor: Fast string matching using an n-gram algorithm. *Software: Practice and Experience*, **24**(1):79–88, 1994. doi:10.1002/spe.4380240105

[KMP77] D.E. Knuth, J.H. Morris, Jr., and V.R. Pratt: Fast pattern matching in strings. *SIAM Journal on Computing* **6**(2):323–350, 1977. doi:10.1137/0206024

[Knu11] D.E. Knuth: *The Art of Computer Programming. Volume 4A / Combinatorical Algorithms, Part 1.* Addison-Wesley, 2011. ISBN-10: 0-201-03804-8. ISBN-13: 978-0-201-03804-0.

[KST03] J. Kytöjoki, L. Salmela and J. Tarhio: Tuning string matching for huge pattern sets. In R. Baeza-Yates, E. Chávez, M. Crochemore, editors, *Proceedings of 14th Annual Symposium on Combinatorial Pattern Matching, CPM '03, Lecture Notes in Computer Science* **2676**:211–224, 2003. doi:10.1007/3-540-44888-8_16

[Lam75] L. Lamport: Multiple byte processing with full-word instructions. *Communications of the ACM*, **18**(8):471–475, 1975. doi:10.1145/360933.360994

[Lec98] T. Lecroq: Experiments on string matching in memory structures. *Software: Practice and Experience*, **28**(5):561–568,

1998. doi:10.1002/(SICI)1097-024X(19980425)28:5<561::AID-SPE170>3.0.CO;2-W

[Lec07] T. Lecroq: Fast exact string matching algorithms. *Information Processing Letters*, **102**(6):229–235, 2007. doi:10.1016/j.ipl.2007.01.002

[Mha05] M.M. Mhashi: The effect of multiple reference characters on detecting matches in string-searching algorithms. *Software: Practice and Experience*, **35**(13):1299–1315, 2005. doi:10.1002/spe.672

[NaR98] G. Navarro and M. Raffinot: A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Proceedings of 9th Annual Symposium on Combinatorial Pattern Matching, CPM '98, Lecture Notes in Computer Science* **1448**:14–33, 1998. doi:10.1007/BFb0030778

[NaR00] G. Navarro and M. Raffinot: Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, **5**(4):1–36, 2000. doi:10.1145/351827.384246

[Nav01] G. Navarro: NR-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, **31**(13):1265–1312, 2001. doi:10.1002/spe.411

[NaR02] G. Navarro and M. Raffinot: *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002. ISBN-10: 0-521-81307-7.
Errata available: `http://www.dcc.uchile.cl/~gnavarro/FPMbook/erratas.html`

[PeT11] H. Peltola and J. Tarhio: Variations of Forward-SBNDM. In *Proceedings of the Prague Stringology Conference 2011*, pages 3–13. Czech Technical University in Prague, Czech Republic. *URL:* `http://www.stringology.org/event/2011/p02.html`

[Rai92] T. Raita: Tuning the Boyer–Moore–Horspool string searching algorithm. *Software: Practice and Experience*, **22**(10):879–884, 1992. doi:10.1002/spe.4380221006

[Rai99] T. Raita: On guards and symbol dependencies in substring search. *Software: Practice and Experience*, **29**(11):931–941, 1999. doi:10.1002/(SICI)1097-024X(199909)29:11<931::AID-SPE264>3.0.CO;2-X

[Sed83] R. Sedgewick: *Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983. ISBN-10: 0-201-06672-6.

[Smy03] B. Smyth: *Computing patterns in strings*. Harlow: Addison-Wesley, 2003. ISBN-10: 0-201-39839-7. Errata available: `http://www.cs.curtin.edu.au/~smyth/patterns.shtml`

[SSA04] S.S. Sheik, S.K. Aggarwal, A. Poddar, N. Balakrishnan, and K. Sekar: A FAST pattern matching algorithm. *J. Chem. Inf. Comput. Sci.*, **44**(4):1251–1256, 2004. doi:10.1021/ci030463z

[Sun90] D.M. Sunday: A very fast substring search algorithm. *Communications of the ACM*, **33**(8):132–142, 1990. doi:10.1145/79173.79184

[TVS06] R. Thathoo, A. Virmani, S.Sai Lakshmi, N. Balakrishnan, and K. Sekar: TVSBS: A fast exact pattern matching algorithm for biological sequences. *Current Science* **91**(1):47–53, 2006.

[War03] H.S. Warren, Jr.: *Hacker's Delight*. First printing, Addison-Wesley, 2003. ISBN-10: 0-201-91465-4. Errata available: `http://www.hackersdelight.org/errata1.pdf`

[WuM91] S. Wu and U. Manber: Fast text searching with errors. Technical report TR 91-11. Department of Computer Science, University of Arizona, 1991.

[WuM92] S. Wu and U. Manber: Fast text searching allowing errors. *Communications of the ACM*, **35**(10):83–91, 1992. doi:10.1145/135239.135244

[WuM94] S. Wu and U. Manber: A fast algorithm for multi-pattern searching, Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.

[ZhT87] R.F. Zhu and T. Takaoka: On improving the average case of the Boyer–Moore string matching algorithm. *Journal of Information Processing*, **10**(3):173–177, 1987.

A. List of algorithms used in test runs

BMH4 — BMH4 is a Boyer–Moore–Horspool type algorithm with the unrolling factor 3 and using 4-grams. It was presented as Algorithm 4 in Publication [I]. This same code was in tests called A4.4 and in Publication [IV] BMH4. (There exist versions also for other q-gram lengths.)

BMH4b — BMH4b is a Boyer–Moore–Horspool type algorithm with the unrolling factor 1. It is using 4-grams and simultaneous 2 byte read. It was presented in Publication [IV].

BR
BRX — BR is our implementation of the Berry–Ravindran algorithm [BeR99]. BRX is our less greedy version, which is described in Publication [IV]. (In Section 3.1 there is some discussion about them.)

EBOM
EBOMb — EBOM (Extended-BOM) is an efficient algorithm based on the oracle automaton [FaL09]. It is utilizing 2-grams. EBOMb is our modification using simultaneous 2 byte read.

Fast-Shift-Or — Fast-Shift-Or (Algorithm 7) is a bit-parallel algorithm introduced in [FrG05]. This implementation uses the unrolling factor 8. Because the speed of Fast-Shift-Or depends quite little on the pattern, this implementation acts also as a reference level of performance.

FAOSO(k,r) FAOSO is a bit-parallel algorithm introduced in [FrG05]. The first parameter k is the sampling rate: every kth text character is processed. (Therefore $k \leq m$.) If a potential match is found, then it is verified. The second parameter r is the unrolling factor. Note that FAOSO(1,*) is like Fast-Shift-Or (Algorithm 7), but makes actually unnecessary verification. This version based on the original implementation expects null-characters in the end of text.

FAOSO uses "integer logarithm", which means $\lfloor \log_2 x \rfloor$. The usual hack is using conversion to a IEEE float and extracting exponent part [War03, p. 215, 83]. This implementation uses Intel's x86 architecture instruction BSR (*Bit Scan Reverse*).

FSB
FSBb
FSBb+ Forward-SBNDM is a lookahead version of the SBNDM2 algorithm. It was introduced by Faro and Lecroq [FaL08, FaL09].

FSB is our implementation of the Forward-SBNDM algorithm. FSBb is a modification using simultaneous 2 byte read. FSBb+ is a modification of FSBb with a tighter skip loop.

HASHq HASH is the implementation of Lecroq's hashing scheme (Section 3.7). The numeral describes the number of text characters used to compute the hash value.

HASH3, HASH4, HASH5, HASH6, HASH7, and HASH8 are based on the original implementations. HASH2c is a modification of HASH3, where the hash value is evaluated to a character variable. HASH2i is a modification of HASH4, where the hash value is evaluated to an integer variable.

KS — Kim and Shawe-Taylor [KiS94] introduced an algorithm for searching in the DNA or RNA alphabet. The alphabet is compressed by masking the three lowest bits of ASCII characters.
KS is an implementation adapted from the original report. It first processes a 5-gram at the end of the current alignment. If that 5-gram does not appear in the end of the pattern, the pattern is moved forward based on the 5-gram. Otherwise two additional characters are checked one by one to achieve a longer shift. Only the prefix of the pattern is matched with the text.

QS — QS is the original implementation `uf.rev.sd1` of Sunday's Quick Search algorithm (Section 2.4). Because this implementation is easily available, QS serves well as a reference level of performance.

SBNDMq
SBNDMqb
SBNDM$q$uf — SBNDMq is an implementation of Algorithm 17. SBNDMqb versions use simultaneous 2 byte read. SBNDM$q$uf is a newer implementation with unrolled fast skip loop. SBNDM2+2b reads first simultaneously the two last text characters from an alignment. If needed, it processes backwards two text characters more, and continues then like SBNDM4b.

Shift-Or — Shift-Or (Algorithm 6) is a bit-parallel algorithm introduced in [BaG92]. The implementation is our fastest version. Because the speed of Shift-Or depends quite little on the data, this implementation acts also as a reference level of performance.

UFNDMq — UFNDMq is an implementation of Algorithm 19.

B. Specifications of the computers used in test runs

Ki (=2^{10}) and Gi (=2^{30}) are prefixes of the IEEE 1541 standard.

A **CPU Model:** AMD Athlon Thunderbird [A4-A8] 1000 MHz (Family: 6 Model: 4 Stepping: 2, Instruction set: x86)

Memory: 256MiB PC133A (SDRAM)

Cache: L1 Instruction cache: 64KiB, 2-way associative. 64 byte line size.

L1 Data cache: 64KiB, 2-way associative. 64 byte line size.

L2 cache: 256KiB, 16-way associative. 64 byte line size.

Motherboard: Asus A7A266

C compiler: gcc (Debian 4.4.5-8) 4.4.5 (configured `--with-arch-32=i586 --with-tune=generic --target=i486-linux-gnu`) with GNU C Library (Debian EGLIBC 2.11.3-4) stable release version 2.11.3 including i686 optimized version 2.11.3

Operating system: Debian GNU/Linux 6.0.6 (squeeze), Kernel: 2.6.32-5-686 #1

B Dell Optiplex 755

CPU Model: Intel Core2 Quad Q9300 2.5GHz (“Yorkfield-6M”), 4 cores, (cpu Mhz: 2000, bus width: 64 bits, Instruction set: x86/x86-64) (Family: 6 Model: 23 Stepping: 7)

Memory: 4GiB DDR2 SDRAM dual symmetric 800 MHz

Cache: L1 Instruction cache: 32KiB, 8-way associative. 64 byte line size.

L1 Data cache: 32KiB, 8-way associative. 64 byte line size.

L2 cache: 6MiB (3MiB/2 cores), 12-way associative. 64 byte line size.

Chipset: Intel Q35 Express (ICH9D0)

C compiler: gcc (Debian 4.4.4-8) 4.4.5 (configured `--with-arch-32=i586 --with-tune=generic --target=x86_64-linux-gnu`) with GNU C Library (Debian EGLIBC 2.11.3-4) stable release version 2.11.3

Operating system: Debian GNU/Linux 6.0.6 (squeeze), Kernel: 2.6.35-5-amd64 #1 SMP

C Dell Optiplex 740

CPU Model: AMD Athlon 64 X2 Dual Core Processor 5000+ 2200MHz (K9 “Windsor”), 2 cores (cpu Mhz: 1000, bus width: 64 bits) (Family: 15 Model: 75 Stepping: 2, Instruction set: x86/x86-64)

Memory: 2GiB DDR2 SDRAM non-ECC memory (667/ 800 MHz) ?

Cache: L1 instruction cache: 128KiB (64KiB/core). 2-way associative. 64 byte line size.

L1 data cache: 128KiB (64KiB/core). 2-way associative. 64 byte line size.

L2 cache: 512KiB (256KiB/core). 16-way associative. 64 byte line size.

Chipset: NVIDIA Quadro NVS 210S with NVIDIA nForce 430 MCP

C compiler: gcc (Ubuntu 4.4.1-4ubuntu9) 4.4.1

Operating system: Ubuntu GNU/Linux 9.10 (karmic), Kernel: 2.6.31-22-generic #68-Ubuntu SMP

This computer was in test runs for Publications [III]–[VI]. After that the computer broke irreparably, and it cannot be used anymore.

D Dell Precision T1500

CPU Model: Intel Core i7-860 2.8GHz, 4 cores, 1 or 2 threads/core (cpu Mhz: 1200, bus width: 64 bits) (Family: 6 Model: 30 Stepping: 5, Instruction set: x86/x86-64)

Memory: 16GiB DDR3 non-ECC memory (1333MHz)

Cache: L1 instruction cache: 8KiB / core. 4-way associative. 32 byte line size.

L1 data cache: 8KiB / core. 8-way associative. 64 byte line size.

L2 cache: 256KiB / core. 8-way associative. 64 byte line size.

L3 cache: 8MiB. 16-way associative. 64 byte line size.

Chipset: Intel P55

C compiler: gcc (Ubuntu / Linaro 4.6.3-1ubuntu5) 4.6.3 (configured `--with-arch-32=i686 --with-tune=generic --target=x86_64-linux-gnu`) with GNU C Library (Ubuntu EGLIBC 2.15-0ubuntu10) stable release version 2.15

Operating system: Ubuntu 12.04 LTS (wheezy/sid) Kernel: 3.2.0-27-generic

H Virtual server running at Sun SPARC Enterprise T5240 server, Sun SPARC V9 architecture

CPU Models: Two 1.165GHz UltraSPARC T2 Plus processors (per system), each having 6 cores, each running at most 8 threads.

Memory: 16GiB (8 x 2GiB) DRAM. Supports 128 bits of write data and 16 bits ECC per SDRAM cycle, and 256 bits of read data and 32 bits ECC per SDRAM cycle. ECC generation, check, correction.

Cache: L1 instruction cache: 16KiB / core. 8-way associative. 32 byte line size.

L1 data cache: 8KiB / core. 4-way associative. 16 byte line size.

L2 cache: shared 4MiB integrated on chip (8-banks), 16-way associative. 64 byte line size.

C compilers: C-compiler: gcc (GCC) 4.4.4 (configured `-mcpu=v9` Target: `sparc-sun-solaris2.10`) and Sun C 5.6 (in hutcs-old)

Operating systems: SunOS 5.8 Generic_Virtual (hutcs-old) and SunOS 5.10 Generic_142900-15

C. Test results with short DNA patterns

The text is the genome of Escheria Coli[1]. Its size is 4,638,690 bytes.

The pattern sets are all the possible combinations of a, c, g, and t for $m = 2, 3, 4, 5, 6$. So the pattern sets contain 16, 64, 256, 1024, and 4096 patterns, respectively.

The preprocessing was repeated 4000 times and the search times are averages of 20 searches. All the implementations were compiled with the gcc compiler (listed in Appendix B). For the 32-bit code the flags `-O3 -m32` were used, and for the 64-bit code the flags `-O3 -m64` were used.[2]

These pattern sets produce an enormous number of matches for each pattern. Fast-Shift-Or seems to suffer a lot of it. The used unrolling factor 8 is the main cause for it. With the shortest patterns, the algorithm goes very often to verification.

[1]The file was from Canterbury Large Corpus
URL: http://corpus.canterbury.ac.nz/descriptions/

[2]Recent blog article 'GCC x86 performance hints' at `http://software.intel.com/en-us/blogs/2012/09/26/gcc-x86-performance-hints` reports about considerable performance improvements with use of some flags. On our tests using those flags and their subsets the changes on exact string matching algorithms were quite small: on some algorithms the performance improved while on some others it degraded.
In gcc version 4.2 a new handy value `native` was introduced to flag `-march`. 'Using `-march=native` enables all instruction subsets supported by the local machine (hence the result might not run on different machines).' Use of this value in speed tests enables risk that some algorithms may get relatively better performance on some CPU model than on other similar computers. It seemed that gcc was able to utilize SSE instructions on few string matching algorithms, while the tuning of floating point arithmetic flags had no effect.

Table C.1. Average preprocessing (leftmost numbers) and search times (after '+' the rightmost numbers) per pattern for short DNA sequences in milliseconds with the 32-bit code on the test computer B.

patterns→ ↓algorithm	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$
SBNDMq1	.000+23.06	.000+16.67	.000+12.76	.000+10.50	.000+ 8.91
SBNDMq2	.000+18.31	.000+12.94	.000+10.00	.000+ 8.78	.000+ 7.79
SBNDMq3	—	.000+15.64	.000+ 8.86	.000+ 6.29	.000+ 5.08
SBNDMq4	—	—	.000+18.82	.000+ 9.80	.000+ 6.68
SBNDMq2b	.033+ 8.88	.033+ 8.58	.033+ 6.74	.033+ 6.20	.033+ 5.63
SBNDMq4b	—	—	.033+ 8.40	.033+ 4.50	.033+ 3.03
UFNDM2	.000+25.69	.000+21.09	.000+18.20	.000+16.02	.000+14.21
UFNDM3	.000+29.75	.000+17.73	.000+14.19	.000+11.92	.000+10.28
UFNDM4	.000+33.31	.000+21.30	.000+15.18	.000+12.35	.000+10.41
UFNDM5	.001+36.63	.000+24.69	.000+18.00	.000+14.17	.000+11.79
FSB	.000+22.50	.000+14.59	.000+11.55	.000+ 9.38	.000+ 7.91
FSBb	.033+20.00	.034+13.66	.034+10.73	.034+ 8.76	.034+ 7.43
FSBb+	.034+18.63	.034+12.84	.034+10.09	.034+ 8.32	.034+ 7.06
FAOSO(1,8)	.001+12.00	.000+ 8.66	.000+ 7.55	.000+ 7.27	.000+ 7.18
FAOSO(1,16)	.000+ 9.75	.001+ 7.09	.000+ 6.04	.000+ 5.73	.000+ 5.66
FAOSO(1,24)	.000+ 8.06	.000+ 6.22	.000+ 5.24	.000+ 4.96	.000+ 4.88
FAOSO(2,8)	.000+35.31	.000+36.30	.000+ 9.78	.000+ 9.83	.000+ 5.61
FAOSO(2,11)	.000+33.50	.000+34.39	.000+ 8.88	.000+ 8.93	.000+ 5.17
FAOSO(2,14)	.000+32.38	.000+33.27	.000+ 8.42	.000+ 8.46	.000+ 4.84
Shift-Or	.000+ 9.50	.000+ 6.98	.000+ 6.23	.000+ 6.07	.000+ 6.04
Fast-Shift-Or	.000+11.75	.000+ 6.16	.000+ 4.60	.000+ 4.24	.000+ 4.14
BR	.083+14.50	.083+12.75	.083+11.04	.083+ 9.85	.083+ 8.88
BRX	.082+13.94	.081+11.39	.081+ 9.53	.081+ 8.13	.081+ 7.19
BMH4	.000+19.75	.000+16.36	.001+ 8.54	.001+ 6.61	.001+ 5.41
BMH4d	.000+19.94	.000+16.34	.109+ 7.29	.109+ 5.61	.109+ 4.61
EBOM	.056+10.44	.056+ 9.25	.056+ 7.64	.056+ 7.28	.057+ 6.79
EBOMb	.057+ 8.88	.057+ 8.05	.057+ 6.82	.057+ 6.60	.058+ 6.29
HASH2c	—	.000+13.42	.000+ 9.43	.000+ 7.28	.000+ 6.05
HASH2i	—	.000+13.50	.000+ 9.36	.000+ 7.33	.000+ 6.07
HASH3	—	—	.000+13.89	.000+ 9.39	.000+ 7.13
HASH4	—	—	—	.000+12.33	.000+ 8.29
QS	.000+16.69	.000+13.84	.000+12.44	.000+11.46	.000+10.84

Table C.2. Average preprocessing (leftmost numbers) and search times (after '+' the rightmost numbers) per pattern for short DNA sequences in milliseconds with the 64-bit code on the test computer B. The variances of search time measurements are given in parentheses below them.

patterns→ ↓algorithm	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$
SBNDMq1	.000+20.84	.000+15.91	.000+12.48	.000+10.37	.000+ 8.82
	(516.6)	(320.5)	(273.7)	(175.9)	(127)
SBNDMq2	.000+13.13	.000+ 9.89	.000+ 7.96	.000+ 7.25	.000+ 6.59
	(75)	(170.9)	(159.6)	(84.9)	(34)
SBNDMq3	—	.000+12.10	.000+ 7.03	.000+ 5.04	.000+ 4.16
		(51.2)	(11.3)	(11.1)	(28.1)
SBNDMq4	—	—	.000+15.17	.000+ 7.94	.000+ 5.43
			(102.5)	(137.8)	(91.7)
SBNDMq2b	.034+ 8.53	.034+ 8.65	.034+ 6.91	.034+ 6.33	.034+ 5.74
	(19.1)	(76)	(127.1)	(93.5)	(61.9)
SBNDMq2+2b	—	—	.034+ 6.57	.034+ 5.71	.034+ 5.07
			(35.6)	(57.4)	(21.3)
SBNDMq4b	—	—	.034+ 8.32	.034+ 4.47	.034+ 3.02
			(107)	(81)	(3.4)
UFNDM2	.001+13.19	.000+12.02	.000+10.87	.000+ 9.67	.000+ 8.61
	(120.6)	(42.6)	(180.8)	(80.5)	(47.2)
UFNDM3	.000+13.97	.000+ 6.88	.000+ 5.81	.000+ 5.09	.000+ 4.50
	(281.6)	(108.7)	(74.7)	(20.9)	(87.5)
UFNDM4	.000+15.22	.000+ 7.88	.000+ 5.09	.000+ 4.23	.000+ 3.61
	(154.7)	(120.9)	(19.6)	(39.3)	(17)
UFNDM5	.000+16.41	.000+ 8.95	.000+ 0.59	.000+ 4.48	.000+ 3.78
	(285.3)	(162)	(96.7)	(83.1)	(42.3)
FSB	.000+23.56	.000+15.33	.000+11.96	.000+ 9.73	.000+ 8.20
	(192.5)	(238.3)	(230.6)	(95.6)	(71.4)
FSBb	.035+19.41	.035+13.70	.035+10.71	.035+ 8.71	.035+ 7.31
	(370.9)	(144.8)	(105.4)	(80.2)	(94.9)
FSBb+	.035+18.56	.034+13.29	.034+10.41	.035+ 8.47	.035+ 7.11
	(107.5)	(193)	(181.6)	(163.7)	(36.5)
Shift-Or	.000+ 7.34	.000+ 7.34	.000+ 7.35	.000+ 7.35	.000+ 7.35
	(99.7)	(99.7)	(101.4)	(101.2)	(101.9)
Fast-Shift-Or	.000+11.38	.000+ 5.92	.000+ 4.39	.000+ 4.01	.000+ 3.92
	(187.5)	(102.5)	(66.9)	(2.5)	(63.5)
BR	.083+14.47	.083+12.56	.083+11.05	.083+ 9.81	.083+ 8.86
	(275.3)	(37.2)	(27.7)	(125.2)	(131.1)
BRX	.083+13.97	.083+11.16	.083+ 9.39	.083+ 8.04	.083+ 7.11
	(260.9)	(74.2)	(149)	(16.6)	(33.7)

Continued on next page

Table C.2 – *Continued from previous page*

patterns→ ↓algorithm	$m = 2$	$m = 3$	$m = 4$	$m = 5$	$m = 6$
BMH4	.000+18.88 (313.8)	.000+15.30 (255.5)	.002+ 9.16 (61.2)	.002+ 7.17 (49.2)	.002+ 5.84 (78.5)
BMH4d	.000+18.88 (323.8)	.000+15.88 (319.9)	.121+ 8.51 (6.5)	.121+ 6.52 (5.9)	.121+ 5.40 (83.2)
EBOM	.017+10.31 (138.8)	.017+ 9.08 (57.2)	.017+ 7.54 (33.4)	.017+ 7.17 (67.9)	.018+ 6.75 (82.4)
EBOMb	.017+ 8.91 (149.1)	.017+ 8.37 (147.1)	.018+ 7.10 (50.9)	.018+ 6.80 (97.8)	.018+ 6.39 (111.3)
HASH2c	—	.000+14.48 (287.5)	.000+10.05 (24.6)	.000+ 7.85 (111.7)	.000+ 6.52 (8.6)
HASH2i	—	.000+14.63 (84.2)	.000+10.16 (69.3)	.000+ 7.96 (146.7)	.000+ 6.60 (28.8)
HASH3	—	—	.000+15.47 (291.5)	.000+10.33 (138.7)	.000+ 7.85 (109.2)
HASH4	—	—	—	.000+14.07 (38.8)	.000+ 9.44 (163.6)
QS	.000+17.63 (116.9)	.000+14.84 (259.8)	.000+13.30 (235.1)	.000+12.29 (241)	.000+11.62 (181.4)

There is quite a lot variation in the search time variances. The trend seems to be that the variances become slowly smaller when the patterns get longer. It is good to remember that the variance is counted from squared differences. Therefore one should summarize them by basing on the standard deviation or using the geometric mean. Surprisingly Shift-Or and Fast-Shift-Or do not have the smallest variances. It is also interesting that QS has a large variance, which slowly increases, when patterns get longer. On the other hand, the stability with SBNDMq and UFNDMq shows as low variances. However, some of their variance values seem to be outliers.

D. Test results with medium length DNA patterns

The text is the genome of Escheria Coli[1]. Its size is 4,638,690 bytes.

The patterns come from test data by Hume and Sunday [HuS91]. They have been extracted from the GenBank DNA database. Every pattern set contains 200 patterns. The sets with 5 and 60 long patterns where taken from longer pattern sets with the `cut` command.

The preprocessing was repeated 3000 times and the search times are averages of 30 searches. All the implementations were compiled with the gcc compiler (listed in Appendix B). For the 32-bit code the flags `-O3 -m32` were used, and for the 64-bit code the flags `-O3 -m64` were used.

The results for all the bit-parallel algorithms are missing the pattern lengths 40, 50 and 60, because in the 32-bit mode the largest integer data type is also 32 bits long.

The longest of the average preprocessing times in the 32-bit mode were: BMH4b 21.9 ms, BR 16.6 ms, BRX 16.3, EBOMb 12.6, EBOM 12.5 ms, FSBb+ 7.0 ms, FSBb 6.8 ms, SBNDMq6b 6.6 ms, SBNDMq4b 6.6 ms, and SBNDMq2b 6.6 ms. The average preprocessing times of all other implementations were less than 0.09 ms. On the other hand, the preprocessing times that are shorter than 1.5 ms are inaccurate and may be biased.

The longest of the average preprocessing times in the 64-bit mode were: BMH4b 24.1 ms, BR 16.6 ms, BRX 16.6, FSB+ 7.0 ms, FSB 7.0 ms, SBNDMq2b 6.8 ms, SBNDMq6b 6.8 ms, SBNDMq4b 6.8 ms, EBOMb 3.9, EBOM 3.8 ms, and KS 3.5 ms. The average preprocessing times of all other implementations were less than 0.2 ms. On the other hand, the preprocessing times that are shorter than 1.5 ms are inaccurate and may be biased. Note that on bit-parallel algorithms the bit-vectors are 64 bits

[1]The file was from Canterbury Large Corpus
URL: http://corpus.canterbury.ac.nz/descriptions/

Table D.1. Average search times of moderate length DNA patterns in milliseconds with the 32-bit code on the test computer B. For each length, the pattern set contains 200 patterns.

	pattern length						
↓algorithm	5	10	20	30	40	50	60
SBNDMq1	2111	1149	636	448	—	—	—
SBNDMq2	1753	1074	615	437	—	—	—
SBNDMq3	1265	639	423	340	—	—	—
SBNDMq4	1960	631	319	245	—	—	—
SBNDMq2b	1235	820	489	353	—	—	—
SBNDMq4b	907	321	213	189	—	—	—
SBNDMq6b	—	499	205	175	—	—	—
SBNDMq2uf	1470	963	584	424	—	—	—
UFNDM3	2396	1377	837	632	—	—	—
UFNDM4	2474	1297	709	—	—	—	—
UFNDM5	2841	1433	740	—	—	—	—
UFNDM6	3278	1645	837	—	—	—	—
FSB	1871	1019	553	390	—	—	—
FSBb	1755	967	529	379	—	—	—
FSBb+	1664	940	523	370	—	—	—
FAOSO(1,8)	1449	1431	1429	—	—	—	—
FAOSO(2,8)	1966	749	—	—	—	—	—
FAOSO(3,7)	7559	900	—	—	—	—	—
FAOSO(4,4)	8225	1728	560	—	—	—	—
FAOSO(5,3)	8133	1701	615	—	—	—	—
FAOSO(6,1)	—	8779	937	635	—	—	—
Shift-Or	1215	1216	1225	1224	—	—	—
Fast-Shift-Or	850	824	823	—	—	—	—
KS	3901	688	297	216	199	186	179
BR	1953	1310	937	794	719	704	666
BRX	1627	1019	696	600	550	529	514
BMH4	1322	645	349	259	217	201	197
BMH4d	1127	556	307	239	215	203	208
EBOM	1450	1061	680	510	412	362	321
EBOMb	1324	991	640	491	401	344	309
HASH3	1883	754	391	297	262	237	225
HASH4	2469	733	333	238	210	199	193
HASH5	—	905	368	253	209	199	200
QS	2225	1829	1777	1762	1804	1785	1757

Table D.2. Average search times of moderate length DNA patterns in milliseconds with the 64-bit code on the test computer B. For each length, the pattern set contains 200 patterns.

	pattern length						
↓algorithm	5	10	20	30	40	50	60
SBNDMq1	2082	1136	619	432	339	282	243
SBNDMq2	1445	973	582	409	326	275	239
SBNDMq3	1013	551	397	333	289	255	229
SBNDMq4	1591	523	280	226	207	195	186
SBNDMq2b	1263	823	489	359	291	248	233
SBNDMq4b	897	321	216	188	177	174	173
SBNDMq6b	—	500	206	178	166	155	149
SBNDMq2uf	1419	933	566	413	332	282	245
UFNDM3	1026	658	450	358	304	260	232
UFNDM4	850	475	293	241	222	206	199
UFNDM5	900	475	272	210	198	187	178
UFNDM6	999	515	289	214	194	184	—
FSB	1943	1053	570	402	320	270	233
FSBb	1741	926	514	374	307	268	244
FSBb+	1693	905	498	364	297	260	233
Shift-Or	1470	1470	1470	1470	1470	1469	1471
Fast-Shift-Or	805	779	780	779	779	779	—
KS	3754	663	290	218	197	190	179
BR	1954	1302	932	790	717	705	669
BRX	1610	1007	697	596	544	528	505
BMH4	1428	703	374	271	227	205	196
BMH4d	1306	650	353	266	233	218	211
EBOM	1426	1047	669	493	410	357	315
EBOMb	1349	996	647	488	406	358	319
HASH3	2068	828	423	318	277	254	235
HASH4	2813	834	375	264	220	207	199
HASH5	—	1001	404	275	222	204	198
QS	2382	1955	1905	1887	1934	1913	1883

long. Therefore the number of bytes that has to be initialized is about two times that with 32-bit bit-vectors.

The search time of UFNDMq is much slower in the 32-bit mode than in the 64-bit mode. The preprocessing of EBOM and EBOMb goes much slower in the 32-bit mode. We have not found any good explanation for these.

E. Test results with patterns of English words

The text is the The King James version of the English Bible[1]. Its size is 4,047,392 bytes.

The patterns are the w.n pattern sets of the test data by Hume and Sunday [HuS91, p. 1224]. Each pattern set contains 200 English words of the same length m. The length varies from 3 to 13. 40% of them are from the whole bible, and the rest from the 1MB test subset.

The preprocessing was repeated 3000 times and the search times are averages of 30 searches. All the implementations were compiled with the gcc compiler (listed in Appendix B). For the 32-bit code the flags `-O3 -m32` were used, and for the 64-bit code the flags `-O3 -m64` were used.

The longest average preprocessing times in the 32-bit mode were: BR 16.6 ms, BRX 16.3, EBOMb 11.6, EBOM 11.5 ms, FSBb+ 6.8 ms, FSBb 6.8 ms, SBNDMq2b 6.6 ms, SBNDMq2+2b 6.6 ms, SBNDMq6b 6.6 ms, and SBNDMq4b 6.6 ms. The average preprocessing times of all other implementations were less than 0.08 ms. On the other hand, the preprocessing times that are shorter than 1.5 ms are inaccurate and may be biased.

The longest average preprocessing times in the 64-bit mode were: BR 16.6 ms, BRX 16.6, FSB 6.9 ms, FSB+ 6.9 ms, SBNDMq2b 6.8 ms, SBNDMq6b 6.8 ms, SBNDMq2+2b 6.8 ms, SBNDMq4b 6.8 ms, EBOM 4.3, and EBOMb 3.6 ms. The average preprocessing times of all other implementations were less than 0.04 ms. On the other hand, the preprocessing times that are shorter than 1.5 ms are inaccurate and may be biased.

The search time of UFNDMq is much slower in the 32-bit mode than in the 64-bit mode. Also the preprocessing of EBOM and EBOMb goes much slower in the 32-bit mode.

[1]The file was from Canterbury Large Corpus
URL: http://corpus.canterbury.ac.nz/descriptions/

Table E.1. Average search times of English word patterns in milliseconds with the 32-bit code on the test computer B. For each length, the pattern set contains 200 patterns.

	pattern length										
↓algorithm	3	4	5	6	7	8	9	10	11	12	13
SBNDMq1	1019	913	847	811	745	691	637	599	561	532	502
SBNDMq2	1369	938	725	597	517	453	405	371	342	321	306
SBNDMq3	2627	1327	901	683	559	469	409	364	329	299	281
SBNDMq4	—	3279	1654	1114	843	681	572	496	439	391	358
SBNDMq2b	463	330	275	239	221	201	195	185	183	179	177
SBNDMq2+2b	—	381	310	265	243	219	206	193	183	181	177
SBNDMq4b	—	1431	733	487	374	311	265	238	215	192	181
SBNDMq6b	—	—	—	2139	1080	711	547	443	371	329	289
SBNDMq2uf	725	511	409	347	313	283	260	241	225	217	211
UFNDM2	2169	1651	1345	1144	1005	898	807	734	685	636	596
UFNDM3	2741	2077	1675	1399	1205	1060	945	851	777	714	665
FSB	941	743	595	517	455	409	369	345	316	301	284
FSBb	612	513	419	380	336	311	285	271	249	243	236
FSBb+	589	480	389	353	316	290	268	247	239	226	219
FAOSO(1,20)	885	883	875	883	879	879	879	888	878	888	888
FAOSO(2,11)	2879	603	599	565	563	561	561	559	559	560	559
FAOSO(3,7)	3184	3107	3162	499	498	500	463	459	461	456	444
FAOSO(4,6)	2519	3213	3271	3211	3205	417	417	421	415	379	380
FAOSO(5,5)	2497	2448	3294	3232	3222	3325	3253	378	378	380	371
FAOSO(6,4)	2486	2439	2475	3378	3372	3484	3402	3386	3419	360	357
Shift-Or	1056	1055	1058	1055	1061	1062	1058	1059	1061	1062	1061
Fast-Shift-Or	725	719	717	716	716	715	715	715	717	717	715
BR	885	770	648	581	521	479	437	410	384	353	342
BRX	1020	835	694	603	539	485	440	405	377	358	329
EBOM	587	425	341	300	271	254	235	220	213	209	200
EBOMb	462	341	279	245	229	212	199	191	191	185	188
HASH2c	1781	1203	919	743	629	547	487	441	398	368	343
HASH2i	1782	1207	918	741	630	547	483	439	400	366	343
HASH3	—	2258	1518	1145	921	776	665	587	526	478	436
QS	1025	826	676	600	541	499	452	432	412	389	373

Table E.2. Average search times of English word patterns in milliseconds with the 64-bit code on the test computer B. For each length, the pattern set contains 200 patterns.

	pattern length										
↓algorithm	3	4	5	6	7	8	9	10	11	12	13
SBNDMq1	911	818	765	741	685	643	597	570	536	512	486
SBNDMq2	869	599	469	392	346	305	280	257	238	226	216
SBNDMq3	1982	1007	681	520	424	359	314	282	257	237	222
SBNDMq4	—	2627	1327	895	679	549	463	402	357	320	294
SBNDMq2b	504	364	304	265	244	224	215	203	199	190	192
SBNDMq2+2b	—	398	322	279	250	231	216	204	198	189	184
SBNDMq4b	—	1570	807	560	429	350	300	264	237	214	204
SBNDMq6b	—	—	—	2266	1181	823	613	507	425	371	330
SBNDMq2uf	726	510	406	345	309	279	258	239	226	214	208
UFNDM2	822	635	528	458	410	371	341	317	294	281	267
UFNDM3	891	671	539	457	396	349	316	289	266	250	238
FSB	1046	819	658	567	497	444	399	370	337	319	301
FSBb	677	566	459	406	370	332	305	269	251	251	242
FSBb+	614	511	415	370	334	302	280	266	251	238	232
Shift-Or	1281	1280	1280	1281	1280	1280	1280	1280	1280	1280	1281
Fast-Shift-Or	687	679	677	676	677	677	678	678	677	677	677
BR	993	872	740	663	598	549	498	466	428	410	383
BRX	1169	957	800	698	624	559	507	463	432	405	381
EBOM	577	416	342	297	273	248	231	217	211	207	199
EBOMb	448	332	285	245	233	214	210	199	192	187	184
HASH2c	2033	1374	1044	844	714	618	549	497	451	413	386
HASH2i	2067	1393	1061	857	725	627	558	505	458	421	391
HASH3	—	2503	1680	1268	1022	857	738	649	580	526	481
QS	1162	929	758	675	610	556	508	482	457	431	410

F. Corrections to the publications

Publication [I]

On the page 853 on second line below the Algorithm 2 the term $m - k$ should be $m - q$.

In the middle of the page 854 'They maintain synonym links, . . . ' should be 'In case of collision they use another hash function'.

Publication [II]

On page 83 the initialization of array $restore$ in Algorithm 2 is here in corrected form as Algorithm 13. The If-clause on line 6 had an error.

Publication [III]

On page 163 the name '*Least cost algorithm*' should be '*Optimal mismatch algorithm*'.

Publication [IV]

On page 420 the line 10 of Algorithm 1 BMHq should be:

10: **if** $k > n$ **then exit**
11: Check the potential occurrence
12: $s \leftarrow r$

Publication [V]

On page 31 the title of **Algorithm 4.3** should be SBNDM2.c (instead of BNDM2.c). Also the sixth line of **Algorithm 4.3** should be `'i += m-2;'` or `'i += m-q;'` (instead of `'i += m-1;'`).

The title of the reference [4] should be ‘An Algorithm for Syntactic Analysis’.

Publication [VI]

The page numbers in the reference [1] should be 295–310.

Aalto-DD 78/2013

ISBN 978-952-60-5156-7
ISBN 978-952-60-5157-4 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science and Engineering
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS