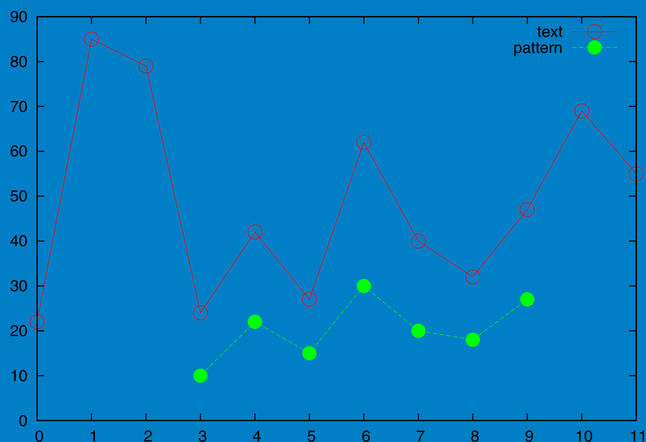
Department of Computer Science

Algorithms for Order-Preserving Matching

Tamanna Chhabra



Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 101/2016

Algorithms for Order-Preserving Matching

Tamanna Chhabra

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 10 June 2016 at 12 o' clock noon.

Aalto University
School of Science
Department of Computer Science

Supervising professor
Professor Jorma Tarhio

Thesis advisor
Professor Jorma Tarhio

Preliminary examiners
Professor Thierry Lecroq, University of Rouen, France
University Lecturer Juha Kärkkäinen, University of Helsinki, Finland

Opponent
Professor Bruce Watson, Stellenbosch University, South Africa

Aalto University publication series
DOCTORAL DISSERTATIONS 101/2016

© Tamanna Chhabra

ISBN 978-952-60-6828-2 (printed)
ISBN 978-952-60-6829-9 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-6829-9

Unigrafia Oy
Helsinki 2016

Finland

NORDIC ECOLABEL
441 697
Printed matter

Aalto University

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

Abstract

Author
Tamanna Chhabra

Name of the doctoral dissertation
Algorithms for Order-Preserving Matching

Publisher School of Science

Unit Department of Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 101/2016

Field of research Software Technology

Manuscript submitted 18 January 2016 **Date of the defence** 10 June 2016

Permission to publish granted (date) 8 March 2016 **Language** English

☐ **Monograph** ☒ **Article dissertation** ☐ **Essay dissertation**

Abstract

String matching is a widely studied problem in Computer Science. There have been many recent developments in this field. One fascinating problem considered lately is the order-preserving matching (OPM) problem. The task is to find all the substrings in the text which have the same length and relative order as the pattern, where the relative order is the numerical order of the numbers in a string. The problem finds its applications in the areas involving time series or series of numbers. More specifically, it is useful for those who are interested in the relative order of the pattern and not in the pattern itself. For example, it can be used by analysts in a stock market to study movements of prices.

In addition to the OPM problem, we also studied its approximate variation. In approximate order-preserving matching, we search for those substrings in the text which have relative order similar to the pattern, i.e., relative order of the pattern matches with at most k mismatches. With respect to applications of order-preserving matching, approximate search is more meaningful than exact search.

We developed various advanced solutions for the problem and its variant. Special emphasis was laid on the practical efficiency of the solutions. Particularly, we introduced a simple solution for the OPM problem using filtration. We proved experimentally that our method was effective and faster than the previous solutions for the problem. In addition, we combined the Single Instruction Multiple Data (SIMD) instruction set architecture with filtration to develop competent solutions which were faster than our previous solution. Moreover, we proposed another efficient solution without filtration using the SIMD architecture. We also presented an offline solution based on the FM-index scheme. Furthermore, we proposed practical solutions for the approximate order-preserving matching problem and one of the solutions was the first sublinear solution on average for the problem.

Keywords string matching, indexing, SIMD, filtration

ISBN (printed) 978-952-60-6828-2 **ISBN (pdf)** 978-952-60-6829-9

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

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2016

Pages 113 **urn** http://urn.fi/URN:ISBN:978-952-60-6829-9

Preface

First, I would like to thank my supervisor, Professor Jorma Tarhio for the support, supervision and guidance he has provided throughout my research. I would also like to thank the former head of department, Professor Heikki Saikkonen and the present head of department Professor Jouko Lampinen, for the funding and all the facilities provided during the research at the Department of Computer Science in Aalto University. I also wish to express my gratitude towards my co-authors M. Oğuzhan Külekci, Simone Faro and Emanuele Giaquinta.

I want to thank Hannu Peltola for his help during my research work. I thank my pre-examiners Juha Kärkkäinen and Thierry Lecroq for reviewing my doctoral dissertation and providing useful comments.

I am also thankful to my mother for the motivation during my research. Finally, I would like to thank my husband Sukhpal Singh Ghuman for all his help and support.

Espoo, May 13, 2016,

Tamanna Chhabra

Contents

Preface 1

Contents 3

List of Publications 5

Author's Contribution 7

1. Introduction 9
- 1.1 Motivation . . . 9
- 1.2 Outline . . . 10

2. Background 13
- 2.1 Terminology . . . 13
- 2.2 Methodologies . . . 14
  - 2.2.1 SIMD Instruction Set . . . 14
  - 2.2.2 Data Structures . . . 19

3. Order-Preserving Matching 21
- 3.1 Definition . . . 21
- 3.2 Solutions by Others . . . 23

4. A Filtration Method for Order-Preserving Matching 29
- 4.1 Solution . . . 29
- 4.2 Analysis . . . 31
- 4.3 Experiments . . . 31

5. Filtering with SIMD and FM-index for Order-Preserving Matching 35
- 5.1 SIMD Approach . . . 35
- 5.2 FM Indexing Approach . . . 39

5.3 Experiments 40

6. SIMD Based Order-Preserving Matching without Filtration 43

6.1 Algorithm 43

6.2 Experiments 47

7. Approximate Order-Preserving Matching with Filtration 49

7.1 Preliminaries 49

7.2 Solutions 50

7.3 Analysis 53

7.4 Experiments 56

8. Conclusions 59

Bibliography 61

Publications 65

Errata 111

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Tamanna Chhabra and Jorma Tarhio. A filtration method for order-preserving matching. *Information Processing Letters*, 116(2): 71–74, 2016.

II Tamanna Chhabra, M. Oguzhan Külekci, and Jorma Tarhio. Alternative algorithms for order-preserving matching. In *Proceedings of the Prague Stringology Conference, Prague, Czech Republic*, 36–46, August 2015.

III Tamanna Chhabra, Simone Faro, and M. Oguzhan Külekci. Engineering order-preserving pattern matching with SIMD parallelism. *Software–Practice and Experience*, 2015 .

IV Tamanna Chhabra, Emanuele Giaquinta, and Jorma Tarhio. Filtration algorithms for approximate order-preserving matching. In *Proceedings of the String Processing and Information Retrieval – 22nd International Symposium, SPIRE*, London, UK, Lecture Notes in Computer Science 9309: 177–187, September 2015.

Author's Contribution

Publication I: "A filtration method for order-preserving matching"

Design of the algorithm was joint work. I implemented the algorithm, performed the experiments, and co-wrote the paper.

Publication II: "Alternative algorithms for order-preserving matching"

I was responsible for the design of the algorithms, implementation of the algorithms, carrying out the experimental results and I wrote the major part of the paper.

Publication III: "Engineering order-preserving pattern matching with SIMD parallelism"

I was involved in the design of the algorithm and I implemented the algorithm, performed the experiments as well as wrote the major part of the paper.

Publication IV: "Filtration algorithms for approximate order-preserving matching"

I co-designed the algorithms, implemented the algorithms, carried out the experimental results and co-wrote the paper.

1. Introduction

1.1 Motivation

Order-preserving matching [3, 12, 37, 39] has gained much attention lately and the thesis is related to the algorithms developed for the problem. The problem finds applications in the fields where we are interested in locating patterns affected by relative orders and not by their values. It can be applied to time series analysis [10] like share prices on stock markets [37], weather data or to musical melody matching of two musical scores [37]. For example, the analysts in a stock market could be interested in the frequency of various movements of the price after a steady decrease of five days. It can also be used by a music composer to find if his new song has a melody similar to any other song. In a similar way, order-preserving matching can be applied to several areas involving series of numbers.

By the year 2014, there were only a few solutions proposed for the problem. Moreover, these solutions were mainly linear and were not much efficient. Therefore, our focus was on the development of faster solutions for the problem.

We have also worked on the approximate variant of the problem and succeeded in developing the first sublinear solution for it. Our algorithms for order-preserving matching and its variant are practical in nature whereas not all algorithms in the literature have ever been implemented.

Our main emphasis is on the practical efficiency of algorithms and could be best described by algorithm engineering [47]. Algorithm engineering is a methodology where design, analysis, implementation and experimental evaluation form a feedback loop driving the development of an efficient algorithm. The loop is traversed until we get a competent algorithm. There-

fore, we show with practical experiments that our new solutions are faster than the previous solutions in most cases. Our solutions are based on the following methodologies:

- *Filtration:* Filtration has proved to be quite effective in our approaches for order-preserving matching, where in we filter out positions in the text which are non-matching.

- *SIMD (Single Instruction Multiple Data):* These instructions were originally developed for multimedia but are recently employed for pattern matching. The general trend in the last decades for speeding up string matching algorithms was based on the word-RAM model, where in several operations on items occupying a single word are assumed to be achieved in constant time. In that context, the advance of the SIMD technology gave rise to packed string matching [4], where one can assume that several consecutive symbols of the underlying text are packed into a single register, and there exist special instructions on those special registers to operate on those items individually. The SIMD instructions were used to create a filter while searching for a long pattern in [40]. This filtration code was listed among the best performing 11 pattern matching algorithms in a recent survey [20]. The same idea was deployed for multiple string matching [16], and then extended to also cover short patterns [17, 18]. Ladra et al. [41] investigated the benefits of using SIMD instructions on compressed data structures, mainly on rank/select operations, and analyzed the Boyer–Moore–Horspool (BMH) algorithm [29] as a case study.

- *FM-index:* We use the FM-index scheme as one of our approach to the order-preserving matching problem. It can be used to count efficiently the occurrences of a pattern in the compressed text and to determine the locations of each pattern in the text.

1.2 Outline

The thesis consists of an overview and the publications. Below I give a

brief summary of papers used in the thesis.

Publication I: A filtration method for order-preserving matching. We present a simple yet efficient algorithm for order-preserving matching. The algorithm is based on filtration and any algorithm for exact string matching can be used as a filtering method. If the filtration algorithm is sublinear, the total method is sublinear on average. We have shown by practical experiments that our solution is more efficient than earlier algorithms in most of the cases. This article is the journal version of a conference paper [11].

Publication II: Alternative algorithms for order-preserving matching. In this paper we introduced two online solutions and an offline solution for the problem. The online solutions are based on two different SIMD (Single Instruction Multiple Data) instruction sets, SSE (streaming SIMD extensions) and AVX (Advanced Vector Extensions). The online solutions use specialized packed string instructions with a low latency and turn out to be faster than the previous online solutions in most cases. The offline solution is based on the FM-index and the execution time decreases substantially for long patterns. We show with practical experiments that one of our solutions is faster than the solutions in Publication I.

Publication III: Engineering order-preserving pattern matching with SIMD parallelism. We proposed a practical and efficient algorithm without filtration for the order-preserving pattern matching problem that turned out to be faster than the best algorithms known. Specialized word-size packed string matching instructions, based on the Intel streaming SIMD extensions (SSE) technology were used. Our experimental results show that the new algorithm is better on average than the algorithms in Publication I and II .

Publication IV: Filtration algorithms for approximate order-preserving matching. We presented two practical solutions for the approximate variant of the order-preserving matching problem. The solutions are based on filtration and their worst-case time complexities are $O(nm(\lceil m/w \rceil + \log m))$ and $O(n(\lceil m/w \rceil \log\log w + m \log m))$, respectively, where w is the word size in bits, and the former is the first sublinear solution on average. We also present experimental results which show that

the filtering is effective and the algorithms are considerably faster than the naive one where all the first $n - m + 1$ text positions are match candidates to be verified.

The rest of the thesis is organized as follows.Chapter 2 contains basic terminology used and the methodologies employed to design the algorithms.

Chapter 3 describes the order-preserving matching problem and its variant, the approximate order-preserving matching problem. We also define both the problems formally. Lastly we describe the previous online and offline solutions for the problem.

Chapter 4 elaborates the explanation of our solution in Publication I with an example. We analyze the solution and perform experimental tests. We also explain how the results are different from the results in Publication I.

Chapter 5 recounts other new algorithms for the problem from Publication II. We explain all the solutions and provide the detailed description using an example. Analysis of the solution is done. Experimental tests are being presented and it is explained how the results differ from the results in the publication.

Chapter 6 describes another efficient solution for order-preserving matching and is based on Publication III. We explain the solution and elucidate it using an example. Further experiments are being performed to show that the new algorithm is faster than the previous solutions in most cases.

Chapter 7 details two new solutions for the approximate order-preserving matching problem and the text is based on Publication IV. We explain the solutions and analyze them. Experimental tests are conducted and then we show how the results in the chapter differ from the results in the publication.

2. Background

2.1 Terminology

A finite, non-empty set of symbols or characters is called an alphabet. It is denoted by Σ. An alphabet may be an English alphabet, an integer alphabet, a binary alphabet and so on. The size of the alphabet is denoted by σ. A string is a finite sequence of symbols over the alphabet Σ. For example, if the alphabet Σ is {a,b}, then ababbb and aaababb are strings on Σ. We suppose that a total order relation "$\leq$" is defined on the alphabet, so that we could establish if $a \leq b$ for each $a, b \in \Sigma$. A substring or factor S' of a string $S = s_0 s_1 \cdots s_{n-1}$ is $S' = s_i \cdots s_j$ where $0 \leq i \leq j < n$. A substring of length q is known as a q-gram. A prefix of the string S is $S' = s_0 s_1 \cdots s_j$ where $j < n$ and suffix of the string is $S' = s_i \cdots s_{n-1}$ where $i \geq 0$. A subsequence of a string S is a string that can be derived from S by deleting some symbols of S, without changing the order of the remaining symbols.

Given a string x, we denote by $|x|$ the length of x and by x_i or $x[i]$ the i-th symbol of x, for $0 \leq i < |x|$ and $x^r = x_{|x|} x_{|x|-1} \ldots x_0$ is the reverse of the string x. The concatenation of two strings x and y is denoted by $x.y$. The Hamming distance between two strings of equal length is defined as the minimum number of substitutions that can transform one string to another. For example, if $x = ababb$ and $y = abbab$, then the Hamming distance between x and y is 2. A bit vector is a binary string $B = b_0 b_1 \ldots b_{n-1}$ such that b_i = 0 or 1. We indicate with symbol w the number of bits in a computer word.

We use some bitwise operations following the standard notation as in C language: $\&$, $|$, $\wedge$, $\sim$, $\ll$, $\gg$ for `and`, `or`, `xor`, `not`, `left shift` and `right shift`, respectively.

Two necessary functions are used in the development of our algorithms explained below. Let x be a string of length m over an alphabet Σ, then:

- *Rank function r:* The rank function of x is a mapping $r : \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$ such that $x[r(i)] \leq x[r(j)]$ holds for each pair $0 \leq i < j < m$. If $x[r(i)] = x[r(i+1)]$ holds, then $r(i) < r(i+1)$.

- *Equality function eq:* Let x be a string of length m over an ordered alphabet Σ and let r be the rank function of x. The equality function of x is a mapping $eq : \{0, 1, \ldots, m-2\} \rightarrow \{0, 1\}$ such that, for each $0 \leq i < m$,

$$eq[i] = \begin{cases} 1 & \text{if} \quad x[r(i)] = x[r(i+1)] \\ 0 & \text{otherwise} \end{cases}$$

2.2 Methodologies

This section explains the methodologies used to develop advanced algorithms for order-preserving matching.

2.2.1 SIMD Instruction Set

SIMD (Single Instruction Multiple Data). In the last two decades a general trend has appeared trying to exploit the power of the word RAM model to speed-up the performance of string matching algorithms. In this model, the computer operates on computer words, grouping blocks of characters and several operations on items occupying a single word are assumed to be achieved in constant time. The SIMD instruction set architecture [34] allows the processor to execute a single instruction on multiple data. For example, one can add several numbers simultaneously in parallel. SIMD instructions were originally used in multimedia and 3D graphics but are recently employed for pattern matching.

Specialized word-size packed string matching instructions, based on the SIMD technology [31, 34] can be employed to design efficient solutions for order-preserving matching. In *packed string matching* [17, 18] sets of adjacent characters are packed into one single word, according to the size of the word in the target machine. This allows us to compare set of characters in a bulk rather than individually, by comparing the corresponding

words. In this case, the symbol w indicates the length of the SIMD register (= 128). Therefore, when the characters are taken from an alphabet of size σ, $\gamma = \lceil \log \sigma \rceil$ bits are used to encode a single character and $\lfloor w/\gamma \rfloor$ characters fit in a register. In this case we will use the symbol $\alpha = \lfloor w/\gamma \rfloor$ to indicate the packing factor.

Recent CPUs manufactured by Intel and AMD support SSE (streaming SIMD extensions) and AVX (Advanced Vector Extensions) instruction sets. The fundamental data types used in the SIMD architecture are bytes, words, doublewords, quadwords, and double quadwords. A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), a quadword is 8 bytes (64 bits), and a double quadword is 16 bytes (128 bits). Besides these fundamental data types, numeric data types such as integer and floating point data types are also supported. Single-precision (32-bit) floating-point and double-precision (64-bit) floating-point data types are also supported by the SSE and AVX instruction set architectures. For the SIMD operation, the data types are either in the packed or scalar form. Packed operations work on several numbers in parallel whereas scalar operations apply an operation on a single value. SIMD programming can be implemented using intrinsic functions. To perform a task, we have various intrinsic functions which depend on the type of instruction set architecture used. An intrinsic function is a function whose implementation is handled specially by the compiler. There are many different versions of SIMD extensions. Three of them are described below:

- *MMX (MultiMedia eXtention):* This instruction set was introduced in the Pentium processor 1997 by Intel. It uses eight 64-bit MMX registers. The limitation of MMX instruction set is that it can handle only integer data. To overcome its limitation SSE (streaming SIMD extensions) was announced in 1999 with the Pentium III processor.

- *SSE (Streaming SIMD Extensions):* It added sixteen new 128-bit registers known as XMM0 through XMM15. However, the registers XMM7–XMM15 are only accessible in the 64-bit operating mode. This architecture supports single-precision floating point operations. As the registers are 128 bits long, it can process four single-precision floating point numbers or two double precision floating point numbers simultaneously thereby providing important speedups in algorithms. In SSE, we have the following data types:

- _m128: four 32-bit floating point values
- _m128d: two 64-bit floating point values
- _m128i: 16/8/4/2 integer values

depending on the size of the integers. Various intrinsic functions are available in SSE to carry out different operations detailed later in this chapter. The identifier of each function starts with a return type. After that follows the descriptive name of the function which describes the operation. The next character specifies whether the operation is on a packed vector or on a scalar value: p stands for a packed and s for a scalar operation. The last character relates to the data type whether it is a single precision or double precision floating point value. Then follows the arguments of the function. For example,

$$__m128\ \ _mm_cmpgt_ps(_m128\ a,\ _m128\ b)$$

is a function for comparing two values, where __m128 is a return type, cmpgt is the descriptive name of the function which describes that it is used to compare two numbers for greater than. The shorthand ps indicates that the function operates on single precision floating point values in a packed form.

- *AVX (Advanced Vector Extensions):* The functionality provided by SSE instructions was extended by Intel AVX (Advanced Vector Extensions) [31]. AVX was first supported by Intel with the Sandy Bridge processor in 2011. It extended the registers to 256 bits known as YMM0–YMM15. Therefore, it becomes possible to process eight single precision floating point numbers or four double precision floating point numbers, simultaneously.

  Various intrinsic functions are used to carry out the implementation of our proposed algorithms. The intrinsic functions used are:

1. Comparison Function (_mm_cmpgt_ps) [32, p. 2A:3-152]: The compiler intrinsic equivalent of the function is

$$__m128\ \ _mm_cmpgt_ps(_m128\ a,\ _m128\ b).$$

A SIMD comparison is performed of the packed single-precision floating-point values in the first operand and the second operand. If a data element in the first operand is greater than the corresponding data element in the second operand, each corresponding data element in the first operand is set to 1, otherwise, it is set to 0.

2. Comparison Function (_mm_cmpgt_epi8) [32, p. 2B:4-81]: The compiler intrinsic equivalent of the function is

$$__m128i\ \ _mm_cmpgt_epi8(_m128i\ a,\ _m128i\ b).$$

It carries out SIMD signed compare of the packed byte (8-bits) in the first operand and the second operand. If a data element in the first operand is greater than the corresponding data element in the second operand, then each corresponding data element in the first operand is set to 1, otherwise, it is set to 0.

3. Comparison Function (_mm_cmpeq_epi8) [32, p. 2B:4-71]: The compiler intrinsic equivalent of the function is

$$__m128i\ \ _mm_cmpeq_epi8(_m128i\ a,\ _m128i\ b).$$

It executes a SIMD compare for equality of the packed byte (8-bits) in the first operand and the second operand. If the corresponding data elements in the first and second operands are equal, then each corresponding data element in the first operand is set to 1, otherwise, it is set to 0.

4. Comparison Function (_mm256_cmp_ps()) [32, p. 2A:3-152]: The compiler intrinsic equivalent of the function is

$$__m256\ \ _mm_cmp_ps(_m256\ a, _m256\ b,\ const\ int\ imm).$$

A SIMD comparison is performed of the packed single-precision floating-point values in the first operand and the second operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a doubleword mask of all ones (comparison true) or all zeros (comparison false) and is stored in the first operand.

5. Mask Function (_mm_movemask_epi8) [32, p. 2B:4-151]: The com-

piler intrinsic equivalent of the function is

$$int\ \ _mm_movemask_epi8(__m128i\ a).$$

It generates a mask made up of the most significant bit of each byte of the source operand (XMM register) and stores the result in the low byte or word of the general purpose register.

6. Mask Function (_mm_movemask_ps) [32, p. 2A:3-568]: The compiler intrinsic equivalent of the function is

$$int\ \ _mm_movemask_ps(__m128\ a).$$

It obtains the sign bits from the packed single-precision floating-point values in the source operand (XMM register) and formats them into a 4-bit mask. The mask is stored in the 4 low-order bits of the general purpose register.

7. Mask Function (_mm256_movemask_ps) [32, p. 2B:3-568]: The compiler intrinsic equivalent of the function is

$$int\ \ _mm256_movemask_ps(__m256\ a).$$

It performs the same function as the above function except that it forms a 8-bit mask.

8. Load Function(_mm_loadu_ps) [32, p. 2A:3-600]: The compiler intrinsic equivalent of the function is

$$__m128\ \ _mm_loadu\ _ps(double * p).$$

It moves four packed single-precision floating-point values from a 128-bit memory location to an XMM register.

9. Load Function(_mm256_loadu_ps) [32, p. 2A:3-600]: The compiler intrinsic equivalent of the function is

$$__m256\ \ _mm256_loadu_ps(__m256 * p).$$

It performs the same function as the above function except that it can move eight packed single-precision floating-point values.

10. Load Function(_mm_loadu_si128) [32, p. 2A:3-551]: The compiler intrinsic equivalent of the function is

$$__m128i \ _mm_loadu_si128(__m128i * p).$$

It moves 128 bits of packed integer values from a 128-bit memory location to an XMM register.

2.2.2 Data Structures

Suffix Arrays. A suffix array (SA) [42] of a string S is an array pointing to the starting positions of the suffixes of S in alphabetical order. Suffix arrays are closely related to suffix trees as a depth-first traversal of the suffix tree yields suffix array. The main advantage of suffix arrays is that they use much less space as compared to suffix trees. A suffix array can be used to quickly locate every occurrence of a pattern P within the string S. It takes $O(m \log n)$ time to locate the pattern P of length m in the string S of length n. But this time can be improved to $O(m + \log n)$ by using the LCP (longest common prefix) information.

Burrows-Wheeler Transform. The Burrows-Wheeler transform (BWT) [6] $BWT[1 \ldots n]$ of a text T is a string of length n such that

$$BWT[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] > 1 \\ \$ & \text{otherwise} \end{cases}$$

where SA is the suffix array of the string T. BWT is obtained by sorting all the rotations of the text into lexicographical order. The last column is then the result of the BWT. BWT is used to permute the string T to $T' = BWT(T)$ and then to reverse T' back to T. It is employed for compression together with run-length encoding and move to front encoding as it produces run of similar characters. The most fascinating property of BWT is the LF (last to first) mapping, which means that the ith occurrence of character x in the last column is the ith occurrence of character x in the first column. This property is employed to unwind the permuted string back to the original string.

Wavelet tree. The wavelet tree of a string on an alphabet Σ of size σ is a data structure to store the string in the form of a bit vector. Each

character of the string is encoded as a bit vector. The root stores the first bits of all the characters of the string. The root's left child stores the second bits of all the characters with a code starting with a 0 and the root's right child stores the second bits of all the characters with a code starting with a 1. The query $rank(i)$ returns the number of ones in the bit vector $B[0 \cdots i]$ and the query $select(j)$ returns the position of jth one in B. The wavelet tree can be used to implement the practical rank and select queries. These operations require $O(\log \sigma)$ time.

FM-Index. Ferragina and Manzini [21] proposed that if BWT [6] is coupled with SA [42], we get a space efficient index which is a sort of compressed suffix array called the FM-index. It can be used to count efficiently the occurrences of a pattern $P[0 \ldots m-1]$ in the text $T[0 \ldots n-1]$ and to determine the locations of each pattern in the text. The operation $count$ takes a pattern and returns the number of occurrences of that pattern in the original text. The BWT of the string is stored as a wavelet tree so that the rank queries can be implemented in $O(\log \sigma)$. Since the rows in BWT of the string are sorted, and it contains every suffix of T, the occurrences of pattern P will be next to each other in a single continuous range. The operation count iterates backwards over the pattern. For every character in the pattern, it finds the range that has the character as a suffix. Therefore, counting the number of occurrences of a pattern P takes $O(m \log \sigma)$ time. The operation $locate$ takes an index of a character in L as an input and returns its position i in the text T. This can be done in $O(m + occ \log^{\epsilon} n)$ time where occ is the number of occurrences.

3. Order-Preserving Matching

The string matching problem [45] is one of the classical problems in computer science. It can be broadly classified as exact and approximate string matching. Exact string matching consists of finding all the occurrences of a pattern string P of length m in a text string T of length n. A natural generalization of the string matching problem can be obtained by allowing the matching to be approximate, so as to search for the substrings in the text T which are *similar* to the pattern P. One classical instance of this kind is the string matching with k mismatches problem, where the task is to find all the substrings in T that are at Hamming distance of at most k from P, i.e., that match P with at most k mismatches.

Over the last few decades, there has been active development in the field of string matching. The problem of order-preserving matching (OPM) [3, 7, 12, 14, 19, 37, 39] has gained much attention recently. This problem considers strings of numbers and has applications in time series studies such as in the analysis of development of share prices in a stock market.

3.1 Definition

The task of order-preserving matching is to find all the substrings u in the text T which have the same length and relative order as the pattern P. By relative order we mean the numerical order of numbers in a string. For example if P is $(12, 19, 15, 8, 10, 24)$ then the relative order of the numbers is 2,4,3,0,1,5. This means that 8 is the smallest number in the string, 10 is the second smallest number, 12 is the third smallest number and so on. Let us assume that $T = (11, 14, 25, 13, 22, 18, 10, 12, 30, 24, 36)$. Now, the substring $u = (13, 22, 18, 2, 8, 30)$ of T has the same relative order as $P = (12, 19, 15, 8, 10, 24)$ and thus P matches T at location 3 as is also shown in Fig. 3.1.

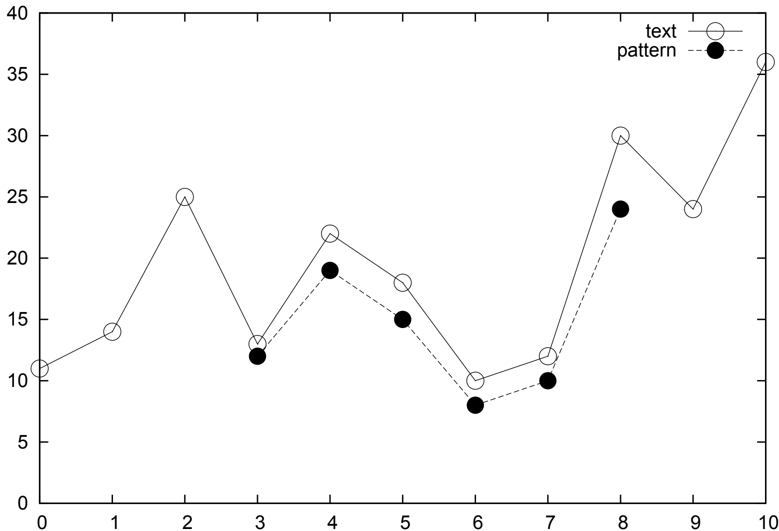


Figure 3.1. Example of order-preserving matching.

As string matching, order-preserving matching can be exact or approximate.

- Exact order-preserving matching: In exact order-preserving matching, the relative order of the pattern P matches substring of the text T exactly. This is an informal description of order-preserving matching and causes problems in handling equal values which can appear in real data. The concept of order-isomorphism removes these problems. Let us define the problem formally.

  Problem definition 1. Two strings $u = u_0 u_1 \dots u_{m-1}$ and $v = v_0 v_1 \dots v_{m-1}$ of the same length over Σ are called *order-isomorphic* [39], written $u \approx v$, if

$$u_i \leq u_j \Leftrightarrow v_i \leq v_j \text{ for } 1 \leq i, j \leq m.$$

  In the *order-preserving pattern matching problem*, the task is to find all the substrings of T which are order-isomorphic with P.

- Approximate order-preserving matching: String matching with k mismatches problem is to find all the substrings of T that are at Ham-

ming distance at most k from P, i.e., that match P with at most k mismatches. Gawrychowski and Uznanski [24] proposed a generalization of the order-preserving matching problem to the approximate case. In this, two strings are k-isomorphic if they have the same relative order after removing up to k elements at the same positions in both strings.

Problem definition 2. Two strings u and v over Σ are *order-isomorphic with k mismatches* [24] or k-isomorphic, written $u \approx_k v$, if they have the same length and there exists a subset K of $\{1, 2, \ldots, |u|\}$ of size k at most, such that

$$u_i \leq u_j \Leftrightarrow v_i \leq v_j \text{ for } i, j \in \{1, 2, \ldots, |u|\} \setminus K \, .$$

The *order-preserving pattern matching with k mismatches* problem is to locate all the substrings in the text T which are k-isomorphic with the pattern P. Let $P = (3, 13, 5, 8, 21)$ and $T = (6, 10, 55, 36, 45, 66, 6, 21, 28, 15, 36)$, then for $k = 1$, we get two approximate matches, at locations 1 and 6.

Several online [3, 7, 12, 19, 37, 39] and one offline solution [14] have been proposed for exact order-preserving matching. But only one solution has been presented for the approximate case [24]. Following section explains all the solutions excluding ours developed until now.

3.2 Solutions by Others

We present our algorithms for order-preserving matching in subsequent chapters. In this section we review solutions made by others.

Kubica et al. [39] proposed the first online solution based on the KMP algorithm [38]. The solution is based on the computation of the order-borders table B where

$$B[1] = 0; B[i] = max\{j < i : P[1 \ldots j] \approx P[i - j + 1 \ldots i]\} \text{ for } i \geq 2$$

where P is a pattern of length m. The table can be computed in linear time. Thereafter it is determined if the text T contains substring with the same relative order as that of the pattern using the order-borders table as shown in Alg. 1. This computation can be done in linear time. Hence, the

total time complexity of the method is linear.

Algorithm 1 Modified algorithm of Morris and Pratt

1: $i \leftarrow 0$
2: $j \leftarrow 0$
3: **while** $i \leq n - m$ **do**
4: $\quad invariant \leftarrow P[1 \cdots j] \approx T[i+1 \cdots i+j]$
5: $\quad$**while** $j < m$ & $P[1 \cdots j+1] \approx T[i+1 \cdots i+j+1]$ **do**
6: $\quad\quad j \leftarrow j+1$
7: $\quad$**end while**
8: $\quad$**if** $j == m$ **then**
9: $\quad\quad$write i
10: $\quad$**end if**
11: $\quad i \leftarrow i + (j - B[j])$
12: $\quad j \leftarrow max(0, B[j])$
13: **end while**

Kim et al. [37] introduced another solution to the order-preserving matching problem based on the KMP algorithm [38]. The solution was based on the natural representation of order relations which means that the numbers in the string are replaced by their ranks in the string. The natural representation can be defined as $\sigma(x) = rank_x(x[1]).rank_x(x[2]).\cdots.rank_x(x[|x|])$, where x is the pattern. The pattern P of length m matches the text T of length n at position i if $\sigma(T[i-m+1 \cdots i]) = \sigma(P)$. But the rank of the number depends on the substring in which it is calculated. Therefore, they further proposed prefix representation in which numbers in the string are replaced by its rank in the prefix. The prefix representation can be described as $\mu(x) = rank_{x1}(x[1]).rank_{x2}(x[2]) \cdots rank_{x|x|}(x[|x|])$. Dynamic order-statistic tree τ was used as the data structure for the construction of prefix representation which could be upgraded incrementally by inserting the next number to it and deleting the previous number from it. The computation of prefix representation is shown in Alg. 2. The time complexity of Compute-Prefix-Rep is $O(m \log m)$ as each of OS-INSERT and OS-RANK takes $O(\log m)$ time and there are $O(m)$ number of such operations.

The KMP failure function in this solution is defined as:

$$\pi[q] = \begin{cases} max\{k : \mu(P[1 \cdots k]) = \mu(P[q-k+1 \cdots q])\} & \text{if } q > 1 \\ 0 & \text{if } q = 1 \end{cases}$$

The failure function π searches the text by filtering mismatched positions in it using the KMP-Order-Matcher [37] as in the KMP algorithm. The prefix representation computation and failure function computation takes

Algorithm 2 Compute-Prefix-Rep(P)

```
1: m ← |P|
2: D ← φ
3: OS-INSERT(τ, P, 1)
4: μ(P)[1] ← 1
5: for k ← 2 to m do
6:     OS-INSERT(τ, P, k)
7:     μ(P)[k] ←OS-RANK(τ, P[k])
8: end for
9: return μ(P)
```

$O(m \log m)$ time whereas text search takes $O(n \log m)$ time. Therefore the total time complexity of the method is $O(n \log m)$.

The prefix representation approach involved an overhead of $O(\log m)$. Therefore, this approach is further optimized using the nearest neighbor representation to overcome the overhead involved in computing the rank function. The main thought behind the approach was to check whether the order of each number in the text matches that of the corresponding number in the pattern by comparing numbers themselves without computing rank values explicitly. The advantage of this method is that without computing rank explicitly we can check whether each number of the text matches the corresponding number of the pattern in constant time. The time complexity of the improved version is $O(n + m \log m)$.

Crochemore et al. [14] proposed an offline solution for the problem. This approach is grounded on the construction of an index that handles the queries in linear time with respect to the length of the pattern. The index is based on the incomplete suffix tree and its construction takes $O(n \log \log n)$ time. They extended their work to complete order-preserving suffix trees and showed how these can be constructed in $O(n \log n / \log \log n)$ time. There exists no practical implementation of this algorithm.

Cho et al. [12] brought forward another solution to order-preserving matching based on the variant of the Boyer–Moore–Horspool (BMH) algorithm [29] built on q-grams, i.e. strings of q numbers. The q-gram version was adopted to make the shifts longer. It uses the shift table D to filter the text so as to achieve sub linear time complexity. The table D is evaluated as follows:

$$k = max\{i \mid P[i-q+1 \cdots i] = x \text{ for } q-1 \leq i \leq m-1\}$$

$$D[f(x)] = min(m - q + 1, m - k - 1)$$

where x is a q-gram, P is the pattern of length m, k is the last position of P matching a q-gram x and D is the shift table. To index the shift table D, they defined a fingerprint $f(x)$ which maps a q-gram x to an integer. In the worst case it takes $O(mn)$ time. Later, Cho et al. [13] introduced a linear version, which has been combined with KMP in order to guarantee linear behavior in the worst case but that is in practice a bit slower than the original one.

Belazzougui et al. [3] presented an optimal sublinear solution for the problem. They viewed the problem in a slightly different way: T is a permutation of $1, \ldots, n$ and P consists of m distinct integers of $[1, n]$. They constructed a forward search automaton working in $O(m^2 \log\log m + n)$ time which is too large for long patterns. With a Morris-Pratt representation [35] of the forward automaton, they achieved $O(m \log\log m + n)$ search time. Furthermore, the automaton was extended to accept a set of patterns. Besides these linear solutions, they presented a sublinear average case algorithm. Firstly, a tree is constructed of all isomorphic order factors of P by inserting factors one at a time. Thereafter search is performed along the text through a window of size m. The construction time of the tree is $O(\frac{m \log m}{\log\log m})$ and average-case time complexity is $O(\frac{n \log m}{m \log\log m})$. However, there exists no implementation of this algorithm so far.

Faro and Külekci [19] presented two filtering approaches in which the original string is translated into a new string over large alphabets. In the neighbourhood ranking approach, a binary sequence of length q is computed which indicates the relative position of the element compared with the elements in its q-neighbourhood. This ordering approach gives information only about the elements in its q-neighbourhood. In the neighbourhood ordering approach, a binary sequence of the element x describes the relative order of the substring $x[i, \ldots, i + q]$.

Cantone et al. [7] later proposed another efficient solution based on the Skip Search algorithm [9]. It computes the fingerprint of all substrings of a pattern of a given length. Thereafter, the fingerprints are indexed to obtain the match candidates which are then located in the text. They used the SSE instruction set architecture for the computation of the fingerprint.

Gawrychowski and Uznanski [24] proposed a solution for approximate order-preserving matching based on the signature of a string. The signature $S(a_0a_1 \ldots a_{m-1})$ of string $a_0a_1 \ldots a_{m-1}$ is $(0 - pred(0), \ldots, (m-1) - pred(m-1))$ where $pred(i)$ is the position where the predecessor of a_i occurs in the string. Its computation takes $O(m \log\log m)$ time by sorting. The key result is that if $a_0a_1 \ldots a_{m-1} \approx_k b_0b_1 \ldots b_{m-1}$ then the Hamming distance between $S(a_0a_1 \ldots a_{m-1})$ and $S(b_0b_1 \ldots b_{m-1})$ is at most $3k$. The algorithm iterates over each substring $t_i \ldots t_{i+m-1}$ in the text T, determining its signature $\mathrm{S}(t_i \ldots t_{i+m-1})$ in $O(\log\log m)$ time per position. For each position i, it checks if the Hamming distance between $S(t_i \ldots t_{i+m-1})$ and $S(p_0 \ldots p_{m-1})$ is greater than $3k$. This step can be done in $O(k + \log\log m)$ time. If the test is true, the position is discarded. Otherwise, the algorithm checks if $t_i \ldots t_{i+m-1} \approx_k p_0 \ldots p_{m-1}$ by reducing the problem to the one of computing a heaviest increasing subsequence [24] spanning at most $3(k+1)$ elements. This step can be computed in $O(k \log\log k)$ time. Therefore, the total time complexity is $O(n(\log\log m + k \log\log k))$.

4. A Filtration Method for Order-Preserving Matching

We present a sublinear solution based on filtration for order-preserving matching. The solution consists of two phases: filtration and verification. For filtration, the pattern and text are transformed into their respective bitmaps where a 1 bit means the successive element is greater than the current one and a 0 bit means the opposite. Then the text is filtered with some exact matching algorithm. The match candidates are then verified using a checking routine. If the filtration algorithm is sublinear, the total method is sublinear on average.

4.1 Solution

Filtration. The consecutive numbers in the pattern $P = p_0p_1 \dots p_{m-1}$ are compared pairwise in the preprocessing phase and as a result we get a transformed pattern $P' = b_0b_1 \cdots b_{m-2}$ as a bit vector, where b_i is 1 if $p_i < p_{i+1}$ holds, otherwise b_i is 0. In the search phase, some algorithm for exact string matching (let us call it A) is applied to filter out the text. When Algorithm A reads an alignment window of the original text, the text is transformed into T' incrementally online in order to skip characters. Algorithm A may recognize an occurrence of P' in T' which does not correspond to an actual match of P in T, and therefore each occurrence of P' in T' is only a match candidate which should be verified.

In simple words, for instance, if $P = (15, 18, 20, 16)$ and $T = (2, 4, 6, 1, 5, 3)$ then P' and T' are 110 and 11010 respectively where 1 indicates an increase and 0 indicates the opposite. P' occurs in T' at location 0 but the relative order of the numbers in the pattern is 0,2,3,1 and the relative order of the numbers in the text at location 0 of the text is 1,2,3,0. Therefore P' is only a match candidate and needs to be verified.

Verification. During preprocessing, the numbers of the pattern $P = p_0 p_1 \dots p_{m-1}$ are sorted. Thereafter the rank function r and equality function eq (described in Chapter 2) for the pattern P are computed. The match candidates found by Algorithm A are traversed in accordance with r. If the candidate starts from t_j in T, the first comparison is done between $t_{j-1+r[0]}$ and $t_{j-1+r[1]}$. There is a mismatch when

$$\begin{aligned} t_{j-1+r[i]} &> t_{j-1+r[i+1]} \text{ or} \\ (t_{j-1+r[i]} &= t_{j-1+r[i+1]} \text{ and } eq[i] = 0) \text{ or} \\ (t_{j-1+r[i]} &< t_{j-1+r[i+1]} \text{ and } eq[i] = 1) \end{aligned}$$

is satisfied. The candidate is discarded when a mismatch is encountered. Verification is efficient because sorting is done only once during preprocessing.

Remark. We use binary numbers in encoding. We also tried encoding of three numbers 0, 1, and 2 corresponding to '$<$', '=', and '$>$', but the binary approach was faster in practice, because testing of one condition is faster than testing of two conditions. Also the frequency of nearby equalities is low in real data.

Example. We illustrate our solution with an example. Let the pattern P be $(10, 22, 15, 30, 20, 18, 27)$ and the text T be $(22, 85, 79, 24, 42, 27, 62, 40, 32, 47, 69, 55, 25)$. The pattern P is transformed into $P' = 101001$ and the text T is transformed into $T' = 100101001100$ incrementally online. Some search algorithm for exact string matching searches for the occurrence of P' in T'. The match candidate of P' is found in T' at location 3 which needs to be verified. For verification the numbers in the pattern need to be sorted. The sorted pattern P_s is $10, 15, 18, 20, 22, 27, 30$. The rank values and equality values for the pattern P are $(0, 2, 5, 4, 1, 6, 3)$ and $(0, 0, 0, 0, 0, 0)$ respectively. The match candidate corresponds to the substring $u = (24, 42, 27, 62, 40, 32, 47)$ of the text T and is traversed in accordance with r. Since the relative order of the numbers of u is the same as the pattern P, a match is found at location 3 of the text.

4.2 Analysis

We will prove that our approach is sublinear in the average case, if the filtration algorithm is sublinear. Sublinearity means that on average all the characters in the text are not examined.

Let us assume that the numbers in P and T are integers and they are statistically independent of each other and the distribution of numbers is discrete uniform. Let P' and T' be the transformed pattern and text. Let c be the count of the integer range (i.e. the alphabet size). The probability of one in a position of P' or T' (as a result of a comparison) is $p = (c^2/2 - c/2)/c^2 = (c-1)/2c$, because there are c^2 integer pairs and c equalities. So the probability q of a character match is

$$p^2 + (1-p)^2 = 2p(p-1) + 1 = 1 - \frac{c-1}{c} \cdot \frac{c+1}{2c} = 1 - \frac{c^2-1}{2c^2} = \frac{1}{2} + \frac{1}{2c^2}.$$

Because adjacent positions in $P' = b_1 b_2 \cdots b_{m-1}$ and in T' are not independent, let us consider matching of a relaxed pattern $P'' = b_1\$b_3\$b_5 \cdots \$b_s$, which contains every other character of P' and where \$ matches both 0 and 1 and s is $2\lfloor m/2 \rfloor - 1$. The probability of a match of P'' at a certain position of T' is smaller than $q^{(m-1)/2}$, which approaches to zero, when m grows. This is true even for $c = 2$. The probability of a match of P' (i.e. a match candidate of P) is smaller than the probability of a match of P''. This means that the verification time approaches zero when m grows, and the filtration time dominates. If the filtration method is sublinear, the total algorithm is sublinear.

The preprocessing phase requires $O(m \log m)$ time due to sorting of the pattern positions. The space requirement is $O(m)$.

In the worst case, the total algorithm requires $O(nm)$ time if, for example, P' is 1^{m-1} and T' is 1^{n-1}. If the filtration method is linear in the worst case, the total algorithm can be modified to work in linear time by combining a linear solution [39, 37] L with it. When the distance of starting positions of subsequent match candidates is less than $m/2$, next $2m$ positions are processed with L.

4.3 Experiments

The tests were run on Intel 2.70 GHz i7 processor with 16 GB of memory running Ubuntu 12.10. All the algorithms were implemented in C and

run in the testing framework of Hume and Sunday [30]. The tests are performed on the time series of relative humidity of UK. The data contains $33,510$ integers representing the relative humidity of UK in percentage in the years 1961–1990. From the text we randomly picked 200 patterns of length $5, 10, 15, 20, 25, 30,$ and 50. Each test was repeated 180 times. The data and testing environment is the same in the subsequent chapters too.

Our solution based on filtration was compared with the BMH approach by Cho et al. [12]. Because the BMH approach was clearly faster than the KMP-based algorithm [37] and slightly faster than the linear version of the BMH approach in the tests [13], we tested only the first mentioned algorithm.

We tested four string matching algorithms as filtration methods for order-preserving matching. Two of them, SBNDM2 and SBNDM4 [15] are based on the Backward Nondeterministic DAWG Matching (BNDM) algorithm [45]. In BNDM, each alignment window is processed from right to left like in the Boyer–Moore algorithm [5] by simulating the nondeterministic automaton of the reversed pattern with bitparallelism. SBNDMq starts the processing of each alignment window by reading a q-gram. The third algorithm is Fast Shift-Or (FSO) [22]. We utilized a version of FSO coded by B. Ďurian [15]. FSO was selected because it is efficient on short binary patterns [15]. The fourth algorithm is the KMP algorithm [38]; together with verification it was supposed to approximate the two earlier methods [37, 39] based on KMP. Of all the algorithms, SBNDM2 and SBNDM4 are sublinear, whereas FSO and KMP are linear.

Table 4.1 shows the average execution times per pattern of all the algorithms in milliseconds. In addition, a graph on times for the data is shown in Fig. 4.2. In the Table, S2OPM represents the algorithm based on SBNDM2 filtration, S4OPM represents the algorithm based on SBNDM4 filtration, BMOPM-q represents the BMH approach [12] for $q = 3, 4$ and 5, KOPM represents the algorithm based on KMP filtration and FSO-OPM represents the algorithm based on Fast Shift-Or.

The results are slightly different from the results presented in Publication I. From Table 4.1, it can be seen that S4OPM is a clear winner for most tested values of m, and FSO-OPM is the fastest for $m = 5$. S2OPM is faster than BMOPM-q and KOPM for all tested values of m and is faster than FSO-OPM for all tested values of m except when $m = 5$. Whereas in Publication I, the execution times of S2OPM and S4OPM for the data sets are comparable and the execution time of S2OPM approaches that of

S4OPM as the value of m increases. The results differ due to the nature of the data. The relative humidity data used in this chapter is more or less stable whereas the data used in the Publication I such as Helsinki temperature data, Dow Jones data and random data portray more variation. The reason for the differences in the experimental results is the same in the subsequent chapters.

Table 4.1. Execution times of algorithms in milliseconds for relative humidity data.

m	KOPM	BMOPM-3	BMOPM-4	BMOPM-5	FSO-OPM	S2OPM	S4OPM
5	39.8	51.5	61.6	127.4	25.6	30.7	44.8
10	38.8	31.0	24.5	27.6	17.2	16.9	13.8
15	39.6	26.9	18.9	16.2	16.4	10.8	5.5
20	40.3	24.6	14.2	12.3	16.3	7.9	4.2
25	40.7	24.0	12.1	10.3	16.3	6.6	4.1
30	39.4	23.0	11.0	8.7	16.6	5.6	3.7
50	39.7	22.8	9.0	6.6	16.6	4.7	3.0

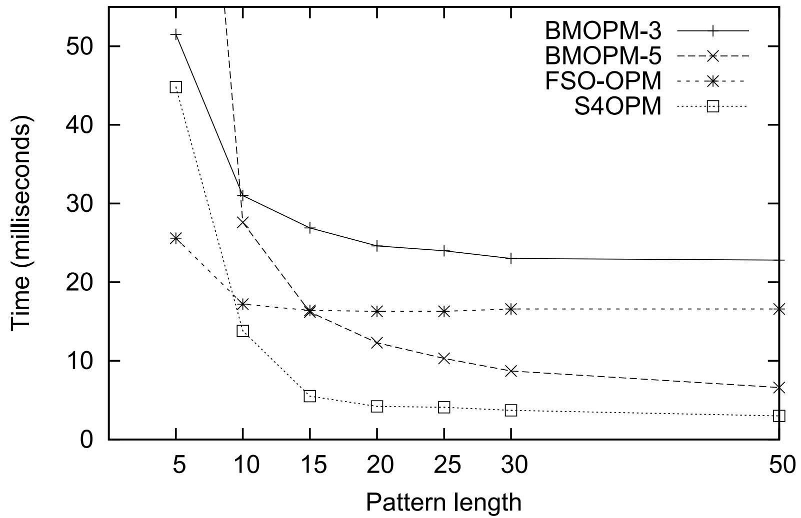


Figure 4.1. Execution times of algorithms for humidity data.

5. Filtering with SIMD and FM-index for Order-Preserving Matching

In this chapter, we introduce three solutions for order-preserving matching, two online solutions and an offline solution, based on filtration. The sublinear average-case solution based on filtration in Chapter 4 is referred to as OPMF, short for order-preserving matching with filtration. All the three solutions are improvements over the OPMF algorithm. In OPMF, pattern P is transformed into P' and text T is transformed incrementally to T'. The online solutions are designed to perform this transformation faster than in OPMF using the SIMD instruction set architecture [34, 32] and are implemented using two different SIMD instruction sets, SSE (streaming SIMD extensions) and AVX (Advanced Vector Extensions) explained in Chapter 2. They use specialized packed string instructions with a low latency and turned out to be faster than the previous online solutions. The offline solution is built on the FM-index scheme [21] described in Chapter 2. The computed bitmap of the text is stored in the compressed form via the FM-index. The transformed pattern is then searched in the FM-index to get potential matches which are then verified. We compare the solutions with OPMF. The experiments show that at least one of the new online solutions is in most cases faster than OPMF. And the indexing solution was the most efficient as one may expect.

5.1 SIMD Approach

This section explains the proposed online solutions for order-preserving matching. The first online solution employs the SSE4.2 instruction set architecture and the second solution utilizes the AVX instruction set architecture.

Online solution using SSE4.2

The online solution based on SSE4.2 for order-preserving matching also consists of two parts: filtration and verification. First the text is filtered and thereafter the match candidates are verified using a checking routine.

Filtration using SSE4.2. We assume that floating point numbers are 32 bits long and the processor has SSE4.2 support. Filtration has two phases, preprocessing and search phase. The preprocessing phase of the pattern consists of two parts. First a bit mask, which is the reverse of P', is formed and thereafter a shift table is constructed based on the mask. For the bit mask, the consecutive numbers in the pattern $P = p_0p_1 \dots p_{m-1}$ are compared pairwise, $(p_0 > p_1)(p_1 > p_2)(p_2 > p_3)\dots(p_{m-2} > p_{m-1})$. This can be achieved by creating $_mm128$ type pointers $ptr1$ and $ptr2$ pointing to p_0 and p_1 respectively. Thereafter, we use the PCMPGT instruction (_mm_cmpgt_ps()) detailed in Chapter 2 to compare $ptr1$ with $ptr2$ to compute $(p_0 > p_1)(p_1 > p_2)(p_2 > p_3)(p_3 > p_4)$ in parallel. The result of this instruction is 128 bits long. Additionally, we use the MOVMSK instruction (_mm128_movemask_ps()) explained in Chapter 2. The reverse of the result is stored in the four low-order bits of the destination operand. The upper bits of the destination operand are filled with zeros. The result is the bit mask *mask*. Alg. 3 shows how the transformation of the pattern P into *mask* can be carried out rapidly.

Since SSE4.2 allows four numbers to be compared in parallel, we apply binary 4-grams and set the size of the shift table *delta* to 16 (= 2^4). The construction algorithm for *delta* is shown in Alg. 4. The computation of the parameter *mask* is explained above. The entry $delta[x]$ is zero if x is the reverse of the last 4-gram of P'. The entries of the table are initialized to $m - 1$. Thereafter, the entries are updated according to the preprocessing in Alg. 4.

Algorithm 3 Transformation of the pattern into a bitmap

1: $mask \leftarrow 0$
2: **for** $i \leftarrow 0$ **to** $m - 1$ **do**
3: $\quad x_ptr \leftarrow _mm_loadu_ps(pattern + i + 1)$
4: $\quad y_ptr \leftarrow _mm_loadu_ps(pattern + i)$
5: $\quad mask \leftarrow mask \,|\, _mm_movemask_ps(_mm_cmpgt_ps(x_ptr, y_ptr)) \ll i$
6: **end for**

The search algorithm shown in Alg. 5 is a variation of the BMH algorithm [29, 48] utilizing 4-grams. Inside the main loop there are two loops. The first loop searches for occurrences of the last 4-gram of P' by using

$delta[0000]$	$\leftarrow 6$				
$delta[0001]$	$\leftarrow 6$				
$delta[0010]$	$\leftarrow 6$			$\leftarrow 3$	
$delta[0011]$	$\leftarrow 6$			$\leftarrow 3$	
$delta[0100]$	$\leftarrow 6$		$\leftarrow 4$		
$delta[0101]$	$\leftarrow 6$		$\leftarrow 4$		
$delta[0110]$	$\leftarrow 6$		$\leftarrow 4$		$\leftarrow 0$
$delta[0111]$	$\leftarrow 6$		$\leftarrow 4$		
$delta[1000]$	$\leftarrow 6$	$\leftarrow 5$			
$delta[1001]$	$\leftarrow 6$	$\leftarrow 5$			
$delta[1010]$	$\leftarrow 6$	$\leftarrow 5$			$\leftarrow 2$
$delta[1011]$	$\leftarrow 6$	$\leftarrow 5$			
$delta[1100]$	$\leftarrow 6$	$\leftarrow 5$			
$delta[1101]$	$\leftarrow 6$	$\leftarrow 5$			$\leftarrow 1$
$delta[1110]$	$\leftarrow 6$	$\leftarrow 5$			
$delta[1111]$	$\leftarrow 6$	$\leftarrow 5$			

Figure 5.1. Computation of the shift table for *mask* = 011010 and $P' = 010110$.

the shift table *delta*. The tested 4-gram is formed online with SIMD instructions in the same way as for the pattern. The numbers are compared in parallel using the PCMPGT instruction explained above (simd-comp in Alg. 5). The second loop checks whether a complete occurrence of P' is found. If an occurrence of P' is found, the corresponding part of T is verified. The search algorithm uses a copy of the pattern as a sentinel (not shown in Alg. 5) to recognize the end of input.

We illustrate the solution using an example. For example, if P is (68,52,66,10,25,36,14) and T is (82,62,43,51,24,33,18,48,72,50,62), then the PCMPGT instruction compares four numbers of the pattern at a stretch, thereby yielding $P' = 010110$ and $mask = 011010$. Thereafter, the shift table *delta* is constructed according to the preprocessing in Alg. 4. Fig. 5.1 shows how the shift table is formed for the pattern P and the entries in it are initialized to 6 as the length of the pattern P is 7 and then it is updated accordingly. At the end, entry 6 is zero. This means that $6 = 0110$ is the reverse of the last 4-gram of P'. Similarly, for the text T, PCMPGT compares four consecutive numbers of it and we get $T' = 0010101101$. The search algorithm shown in Alg. 5 finds the occurrence of P' within T' and then the corresponding part of T at location 4 is verified using a checking routine.

Verification. The verification process is the same as in OPMF. In the preprocessing phase, the numbers of the pattern $P = p_0 p_1 \cdots p_{m-1}$ are sorted. Thereafter the rank function r and equality function eq (described in Chapter 2) for the pattern P are computed. The potential candidates obtained from the filtration phase are traversed in accordance with r. If the candidate starts from t_j in T, the first comparison is done between $t_{j+r[0]}$ and $t_{j+r[1]}$.

Algorithm 4 Preprocessing (mask)

```
1: for i ← 0 to 15 do
2:    delta[i] ← m − 1
3: end for
4: k ← (mask ≪ 3) & 0xf
5: for i ← 0 to 7 do
6:    delta[k + i] ← m − 2
7: end for
8: k ← (mask ≪ 2) & 0xf
9: for i ← 0 to 3 do
10:   delta[k + i] ← m − 3
11: end for
12: k ← (mask ≪ 1) & 0xf
13: for i ← 0 to 1 do
14:   delta[k + i] ← m − 4
15: end for
16: for i ← 0 to m − 3 do
17:   delta[(mask ≫ i) & 0xf] ← m − i − 5
18: end for
```

Algorithm 5 Search($Text, delta$)

```
1: i ← m − 5
2: while i < n do
3:    k ← 1
4:    while k > 0 do
5:       k ← delta[simd-comp(t_i, t_{i+1}, 4)]
6:       i ← i + k
7:       for j ← i − m + 5 to i  step 4 do
8:          z ← simd-comp(t_j, t_{j+1}, 4)
9:          if  z ≠ ((mask ≫ (j − i + m − 5)) & 0xf))  then
10:            goto out
11:         end if
12:      end for
13:      verify occurrence
14:      out : i ← i + 1
15:   end while
16: end while
```

Online solution using AVX

The AVX solution is similar to the above solution with a few exceptions. The difference is in the comparison of numbers and in computation of the shift function. Instead of four numbers, eight floating point numbers are compared at a stretch. The comparison instruction used is _mm256_cmp_ps() and the mask is computed using _mm256_movemask_ps() explained in Chapter 2.

Analysis

Our SIMD search algorithm is a variation of 4-gram BMH. A binary 4-gram corresponds to a character of an alphabet of 16 ($= \sigma'$) characters. Baeza-Yates and Régnier [2] show that in this alphabet, the average value of shift for the traditional BMH is at least $(\sigma' + 1)/2 = 8.5$ when $m \geq \sigma'$ holds and the distribution of characters is discrete uniform. This is roughly true also for our algorithm and the average value of shift approaches to 16 when m grows (we skip the formal proof). Because the verification time approaches zero when m grows and the text is encoded incrementally, the total algorithm is sublinear for $m \geq \sigma'$ on average. In the worst case the algorithm requires $O(nm)$ time as OPMF. The preprocessing phase requires $O(m \log m)$ time due to sorting of the pattern positions.

5.2 FM Indexing Approach

In the FM indexing approach, the bitmaps of text T and pattern P are also enumerated but the bitmap T' of text T is stored in the compressed form via the FM-index. When a pattern is queried, we just extract the possible candidate positions from the index, and then apply naive check. It also consists of two parts: filtration and verification.

Filtration. In the preprocessing phase, the consecutive numbers in the pattern $P = p_0 p_1 \ldots p_{m-1}$ are compared pairwise and the pattern P is transformed into a bitmap P' in the same way as in OPMF. The text is also encoded and an FM-index is created of the encoded text. Alg. 6 below shows how the encoded text is stored in the form of an FM-index. Thereafter, the occurrences of the transformed pattern P' are found within the compressed text. As an occurrence of P' is only a potential match candi-

date, it should be verified with a checking routine.

Verification. The verification process is the same as in the online solution because once we get the potential matches they are verified using the same checking function.

Algorithm 6 FM-index

1: $std :: stringstr((char*)\ \&\ text[0], n)$
2: $construct_im(fm_index, str.c_str(), 1)$
3: $matches \leftarrow count(fm_index, (const\ char*)P')$
4: $auto\ locations \leftarrow locate(fm_index, (const\ char*)P'$

Analysis

Let us assume that the numbers in $P = p_0p_1 \cdots p_{m-1}$ and $T = t_0t_1 \cdots t_{n-1}$ are integers and they are statistically independent of each other and the distribution of numbers is discrete uniform. In the case of the offline solution using FM-index, the verification time approaches zero when m grows and the filtration time dominates. During the preprocessing phase, the bitmap T' is compressed and stored via the FM-index. The operation *count* takes a pattern P' and returns the number of occurrences of that pattern in the text T'. It can count all matching positions in $O(m)$ time. The operation *locate* finds the locations of all the occurrences (occ) of the pattern P' in T' in time $O(m + occ \log^{\epsilon} n)$. However, this solution also requires $O(nm)$ time in the worst case because checking a match candidate takes $O(m)$ time.

5.3 Experiments

The experimental setting is the same as is described in Section 4.3. We compared the solutions with our OPMF solutions (based on SBNDM2 and SBNDM4) detailed in Chapter 4. Table 5.1 shows the average execution times of the algorithms for relative humidity data in milliseconds. In addition, graph on times for the data is also shown in Fig. 5.2. In Table 5.1, SBNDM2 represents the OPM algorithm based on SBNDM2 filtration, SBNDM4 represents the OPM algorithm based on SBNDM4 filtration, SSE represents the online solution based on the SSE4.2 instruction set, AVX represents the online solution based on the AVX instruction set and FM-INDEX represents the offline solution based on the FM index. The FM-index was implemented using the sdsl library [26].

Table 5.1. Execution times of algorithms in milliseconds for relative humidity data.

m	SBNDM2	SBNDM4	SSE	AVX	FM-INDEX
5	22.4	32.8	24.3	—	271.9
10	12.6	10.2	10.2	9.3	29.4
15	7.9	5.4	6.9	5.6	5.2
20	5.8	4.2	5.8	4.4	2.1
25	4.8	4.1	5.2	3.8	1.3
30	4.2	3.6	4.8	3.4	1.1
50	3.4	3.0	3.4	1.9	0.8

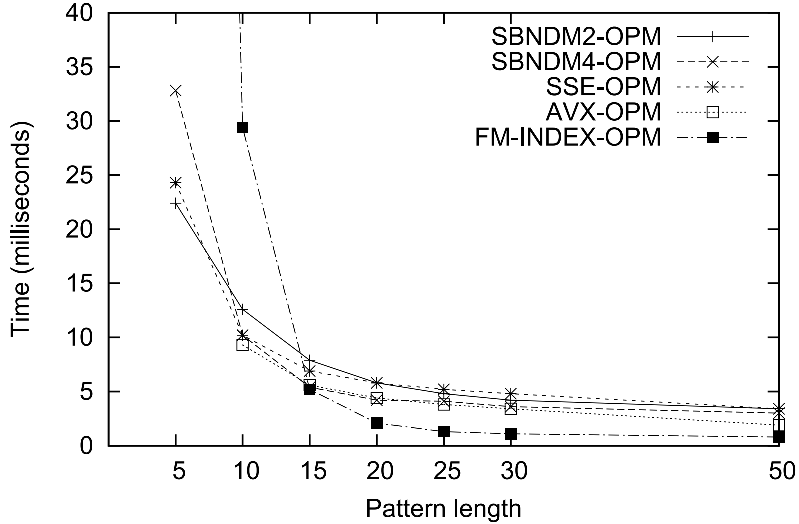


Figure 5.2. Execution times of algorithms for humidity data

The results are different from the results presented in Publication II. From Table 5.1, it can be clearly seen that our solutions based on the FM-index, SSE4.2 and AVX are the fastest depending on the value of m except for $m = 5$. However, irrespective of the data sets tested in Publication II, the solution based on SSE4.2 is the fastest for $m = 5$. As the value of m reaches 10 in the humidity data set, the AVX solution becomes the fastest. However, when m is greater than or equal to 15, the FM-index based solution is the fastest. And as the value of m reaches 50, the execution time of FM-index based solution approaches zero. But in case of Dow Jones data in Publication II, FM-index based solution is the fastest as the value of m reaches 10. The construction times of the FM-index for relative humidity data is 0.01 seconds and construction times of the FM-index for Dow Jones and random texts in Publication II were 0.07 and 3.2 seconds, respectively.

The FM-index based solution is very slow when the value of m is small.

It is due to cache inefficiency. For small values of m, the number of match candidates is large and all the candidates are validated almost randomly and thus the locality of a reference is lost. It is similar to randomly walking in the text. So the FM-index behaves much worse than the online solutions due to cache misses. Another possible reason for the slowness of FM-index for short patterns is that locate is a slow operation in FM-index. A short pattern produces a lot of candidates that have to be located to be verified [36]. However, when the pattern becomes longer, the number of candidates decreases significantly and the FM-index becomes advantageous.

6. SIMD Based Order-Preserving Matching without Filtration

This chapter focuses on another practical and efficient algorithm without filtration for the order-preserving matching problem. We use specialized word-size packed string matching instructions, based on the SSE technology [31, 34] discussed in Chapter 2, to design a very fast order-preserving matching algorithm. The algorithm is named SIMD-OPM and turns out to be faster than the online solutions in Chapter 4 and 5.

Several values of α and γ (explained in Chapter 2) are possible but we assume that $\alpha = 16$ and $\gamma = 8$, which is the most common case when we deal with a word RAM model with 128-bit registers. In our experimental evaluation (see Section 6.2) we have $\sigma = 256$.

We will also make use of the $\mathsf{popcount}(C)$ instruction, when we will be interested in counting the number of bits set in an α-bit register C. This can be done in $\log(\alpha)$ operations by using a population count function. In our implementation we make use of a constant time ad-hoc procedure [1] designed to work with 16-bit registers.

6.1 Algorithm

The SIMD-OPM algorithm is designed to search order-preserving occurrences of a pattern in a text.

Let P be the pattern of length m over the alphabet Σ and if Y is a block of w bits (α elements) of the text T, we can find all the occurrences of P having their leftmost position in Y.

Let $T = Y_0 Y_1 \ldots Y_{k-1}$, where $k = \lfloor n/\alpha \rfloor$+1. The idea behind the algorithm is to check in parallel for groups of occurrences of P in T while scanning each block Y_i of the text. In particular for each iteration of the algorithm we check groups of α occurrences of P.

The SIMD-OPM algorithm makes use of the wspc (*word-size parallel*

comparison) and wsec (*word-size equality checker*) specialized word-size packed instructions. These two instructions are described below.

The instruction wspc

The instruction wspc(A, B), handles two w-bit registers B and A as a block of α small integer values and computes an α-bit fingerprint from it. It compares in parallel all the α values contained in A against the α values in B. More formally, assuming $B[0 \ldots \alpha - 1]$ and $A[0 \ldots \alpha - 1]$ are w-bit integer parameters, wspc(A, B) returns an α-bit value $r[0 \ldots \alpha - 1]$, where $r[j] = 1$ if and only if $A[j] < B[j]$, and $r[j] = 0$ otherwise.

The wspc(A, B) instruction uses the following sequence of specialized SIMD instructions and can be completed in constant time:

```
wspc(A, B)
B ← _mm_cmpgt_epi8(B, A)
r ← _mm_movemask_epi8(B)
return r
```

The instruction wsec

The instruction wsec(A, B), handles two w-bit registers A and B as a block of α small integer values and computes an α-bit fingerprint from it. Assuming $A[0 \ldots \alpha - 1]$ and $B[0 \ldots \alpha - 1]$ are w-bit integer parameters, wsec(A, B) returns an α-bit value $r[0 \ldots \alpha - 1]$, where $r[j] = 1$ if and only if $A[j] = B[j]$, and $r[j] = 0$ otherwise.

The wsec(A, B) instruction uses the following sequence of specialized SIMD instructions and can also be completed in constant time:

```
wsec(A, B)
B ← _mm_cmpeq_epi8(A, B)
r ← _mm_movemask_epi8(B)
return r
```

The instructions _mm_cmpgt_epi8, _mm_cmpeq_epi8 and _mm_movemask_epi8 are described in Chapter 2.

Formally, let $Y_i = T[i\alpha \ldots i\alpha + \alpha - 1]$ be the current block of the text. The substring $T[j \ldots j + m - 1]$ is an order preserving occurrence of P if and only if

- $T[j + r(h)] \leq T[j + r(h + 1)]$, for $0 \leq h < m - 1$

- $T[j + r(h)] = T[j + r(h + 1)]$ if and only if $eq(h) = 1$, for $0 \leq h < m - 1$

Algorithm 7 SIMD-OPM(P, m, T, n)

```
1:  k ← 0
2:  for i ← 0 to n − m, step α do
3:      C ← 1^α
4:      for j ← 0 to m − 2 do
5:          A ← T[i + r(j) . . . i + r(j) + α − 1]
6:          B ← T[i + r(j + 1) . . . i + r(j + 1) + α − 1]
7:          if eq(j) then
8:              C ← C and wsec(A, B)
9:          else
10:             C ← C and wspc(A, B)
11:         end if
12:         if C = 0 then
13:             goto out
14:         end if
15:     end for
16:     k ← k + popcount(C)
17:     out:
18: end for
19: return k
```

The pseudocode of the SIMD-OPM algorithm is given in Alg. 7. During each iteration the algorithm checks the occurrences whose first position is in the block $Y = T[i \ldots i + \alpha - 1]$. At the end of the iteration the value of i is advanced α positions to the right. Thus the total number of iterations of the algorithm is $\lceil n/\alpha \rceil$. The blocks containing positions of $T[i], i = n - m + 1, \cdots, n - 1$, should be processed in another way because there is a possibility of false matches.

During each iteration the algorithm creates a bit mask C of α bits, which contains occurrences of the pattern in the current block Y. Specifically at the end of the iteration the bit $C[j]$ is set if and only if $P \approx T[j \ldots j+m-1]$, for $j = 0 \ldots \alpha - 1$, while $C[i] = 0$ otherwise. At the beginning of each iteration C is initialized as 1^α (line 3).

In order to understand how such a value is computed, let $A_j = y[i + r(j) \ldots i+r(j)+\alpha-1]$ (line 5) and $B_j = y[i+r(j+1) \ldots i+r(j+1)+\alpha-1]$ (line 6). Moreover let $C_j = \mathsf{wspc}(A_j, B_j)$ (line 10). According to the definition of the wspc instruction, we have $C[h] = 1$ if and only if $A[h] < B[h]$ (i.e. $T[i+h+r(j)] < T[i+h+r(j+1)]$), and $C[h] = 0$ otherwise, for $h = 0 \ldots \alpha-1$.

The value of the bit mask C is computed as $C = C_0 \& C_1 \& \ldots \& C_{m-2}$. It is

easy to prove that $C[h]$ is set if and only if $T[i+h+r(j)] < T[i+h+r(j+1)]$ for $j = 0 \dots m-2$, which implies that $P \approx T[i+h \dots i+h+m-1]$. Observe that, when $eq(j) = 1$, we compute C_j as $\mathsf{wsec}(A_j, B_j)$ (lines 7-8) in order to test whenever $T[i+h+r(j)] = T[i+h+r(j+1)]$.

At the end of each iteration we count the number of bits set in the bit mask C. This is the number of occurrences the algorithm found in the current block. Such a value is accumulated in a counter k (line 16) which will contain the total number of occurrences of P in T.

If we are also interested in retrieving the position of each occurrence, an additional $\mathcal{O}(\log \alpha)$ job must be done in order to locate the bits set in C. More specifically if the h-th bit of C is set than an occurrence at position $i + h$ must be reported. The total time complexity of the algorithm is $\mathcal{O}(nm/\alpha)$.

We illustrate the algorithm using an example. Let $P = (8, 5, 13, 10)$ be a pattern of length 4 and $T = (7, 9, 5, 14, 13, 22, 16, 10, 3, 13, 11, 10, 11, 8, 9, 2)$ be a text of length 16. The rank values and equality values for the pattern P are $(1, 0, 3, 2)$ and $(0, 0, 0)$, where $Y = T$. It involves $m - 1 = 3$ steps. We know that $P[r(i)] \leq P[r(i+1)]$, where $0 \leq i \leq m-1$. In order to have an occurrence beginning at position j of Y we must have $Y[j + r(i)] \leq Y[j + r(i+1)]$, for $0 \leq i \leq m-1$. Then C_i is a 16-bit register where $C_i[j]$ is set to 1 if $Y[j + r(i)] \leq Y[j + r(i+1)]$ and $C_i[j]$ is set to 0 otherwise.

Now, C is a 16-bit register where $C = C_0 \ AND \ C_1 \ AND \ \cdots AND \ C_{m-2}$ and $C[j]$ is set if we have an occurrence of P at position j of Y and $C[j] = 0$ otherwise.

Step 1 is as follows:

Y<<1	9	5	14	13	22	16	10	3	13	11	10	11	8	9	2	0
Y<<0	7	9	5	14	13	22	16	10	3	13	11	10	11	8	9	2
C_0	0	1	0	1	0	1	1	1	0	1	1	0	1	0	1	1

Step 2:

Y<<0	7	9	5	14	13	22	16	10	3	13	11	10	11	8	9	2
Y<<3	14	13	22	16	10	3	13	11	10	11	8	9	2	0	0	0
C_1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0

Step 3:

Y<<3	14	13	22	16	10	3	13	11	10	11	8	9	2	0	0	0
Y<<2	5	14	13	22	16	10	3	13	11	10	11	8	9	2	0	0
C_2	0	1	0	1	1	1	0	1	1	0	1	0	1	1	0	0

Then we can compute the value of C as follows.

C_0	0	1	0	1	0	1	1	1	0	1	1	0	1	0	1	1	AND
C_1	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	AND
C_2	0	1	0	1	1	1	0	1	1	0	1	0	1	1	0	0	AND
C	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	=

Thus we found three occurrences of P in T. The first at position 1, the second at position 3 and the last at position 7.

6.2 Experiments

This section presents experimental results in order to compare the behavior of the SIMD-OPM algorithm against the best known solutions in the literature for the OPM problem.

The tests were run on an Intel 2.70 GHz i7 processor running Ubuntu 12.10 with 16 GB of memory. All the algorithms were implemented using C programming language and run in the testing framework of Hume and Sunday [30].

We tested our algorithm SIMD-OPM against the most effective previous solutions which include S2OPM and S4OPM (detailed in Chapter 4), SSEOPM and AVXOPM (explained in Chapter 5), FFK-OPM [19] and SKIP-OPM [7]. S2OPM and S4OPM (given in Chapter 4) solutions are

based on SBNDM2 and SBNDM4 [15]. SSEOPM and AVXOPM represent the online solution grounded on SSE4.2 and AVX instruction set respectively. FFK-OPM [19] presents the filtration approach by Faro and Külekci. SKIP-OPM [7] represents the solution based on Skip Search algorithm.

Table 6.1 shows the average execution times per pattern of all the algorithms for the humidity data in milliseconds. A graph of times for the data set is also shown in Fig. 6.1.

From the table, we can observe that SIMD-OPM is fastest for all the tested values of m except for $m = 50$. The difference between the execution times of SIMD-OPM and other solutions is the maximum when $m = 5$ and thereafter the difference drops. We notice that our algorithm shows a linear behavior.

Table 6.1. Execution times of algorithms in milliseconds for relative humidity data.

m	S2OPM	S4OPM	SSEOPM	AVXOPM	FFK-OPM	SKIP-OPM	SIMD-OPM
5	26.3	43.5	38.7	—	29.5	28.7	4.0
10	14.2	14.9	15.1	12.6	24.1	24.9	4.1
15	9.2	9.3	9.9	7.5	16.3	15.8	3.9
20	6.8	6.5	8.3	6.0	12.8	13.7	3.9
25	5.3	5.0	7.4	5.2	10.9	12.0	3.8
30	4.2	3.9	6.9	4.5	9.6	11.1	3.8
50	3.2	3.0	4.8	2.6	7.1	9.1	3.8

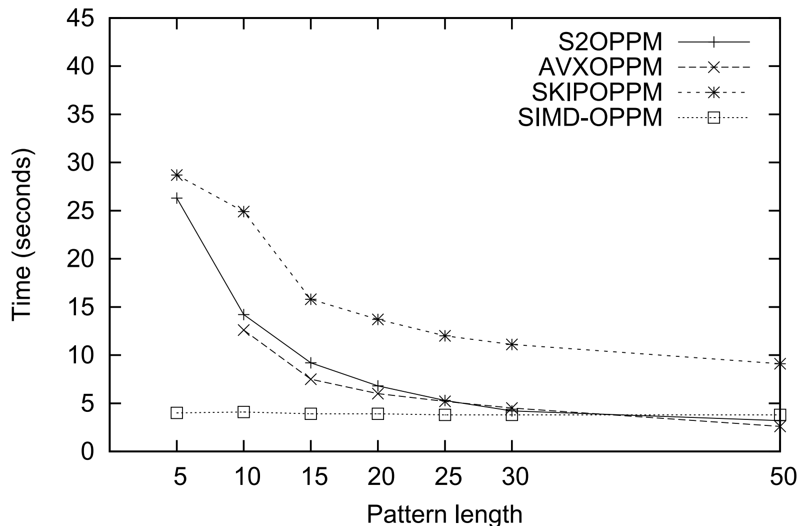


Figure 6.1. Execution times of the algorithms for humidity data.

7. Approximate Order-Preserving Matching with Filtration

In this chapter, we consider the approximate variant of order-preserving matching discussed in Chapter 3. In approximate order-preserving matching, two strings match if they have the same relative order after removing up to k elements at the same positions in both the strings. We introduce two practical solutions for the approximate order preserving matching problem, based on filtration. Their worst-case time complexities are $O(nm(\lceil m/w \rceil + \log m))$ and $O(n(\lceil m/w \rceil \log\log w + m \log m))$, respectively, where w is the word size in bits, and the former is the first sublinear solution on average. We also performed experimental tests which show that the filtering is effective and the algorithms are considerably faster than the naive solution where all the first $n - m + 1$ text positions are match candidates to be verified.

With respect to applications of order-preserving matching (see Chapter 3), approximate search is more meaningful than exact search. Gawrychowski and Uznanski [24] defined the approximate order-preserving matching problem and presented a solution for it (explained in Chapter 3). The idea in their method is to quickly filter out positions in the text T which are non-matching by comparing signatures of the pattern and of the text substrings. As also acknowledged by the authors, this algorithm is rather theoretical and has not been implemented to date.

7.1 Preliminaries

In this chapter the notation used for the transformation is different and is described as follows. Given a string u, we denote by $\phi(u)$ the binary string of length $|u| - 1$ such that $\phi(u)_i$ is equal to 1, if $u_i < u_{i+1}$, and to 0 otherwise. The function ϕ is a linear approximation of the order for fast filtration. Observe that any position $2 \leq i < |u|$ in u covers two positions

in $\phi(u)$, $i-1$ and i. Let u and v be two strings and consider the mismatches between the strings as $\phi(u)$ and $\phi(v)$. Each mismatch position i identifies a different relative order, in u and v, between the adjacent symbols at positions i and $i+1$.

Given a string x and a permutation π of $\{1, 2, \ldots, |x|\}$ we denote by $\pi(x)$ the string $x_{\pi(1)}x_{\pi(2)} \ldots x_{\pi(|x|)}$. Given two strings x and y of length m, the Hamming distance between x and y is $d_h(x, y) = |\{0 \leq i < m \mid x_i \neq y_i\}|$, and the matching statistics $M(x, y)$ is an array of $|x|$ integers where $M(x, y)[i]$ denotes the length of the longest substring of x starting at position i that exactly matches a substring of y. We denote by $H(x, y)$ the largest subset of the mismatch positions between x and y such that no two positions are consecutive, and define a distance measure $d_o(x, y) = |H(x, y)|$. Therefore, for any two strings u and v, there is no overlap between the positions in u and v covered by any two mismatches in $H(\phi(u), \phi(v))$. For any two strings u and v such that $u \approx_k v$ (order-isomorphic with k mismatches explained in Chapter 3), the Hamming distance between $\phi(u)$ and $\phi(v)$ is at most $2k$ i.e. $d_h(\phi(u), \phi(v)) \leq 2k$ and distance measure $d_o(\phi(u), \phi(v)) \leq k$ (see Lemma 1 and 2 in Publication IV).

7.2 Solutions

Our solutions for approximate order-preserving matching consist of two parts: filtration and verification. First the text is filtered with an algorithm so as to locate all the potential matching locations and then the match candidates are verified using a checking routine.

Filtration. The consecutive numbers in the pattern P are compared pairwise in the preprocessing phase and transformed into the binary string $\phi(P)$ where a 1 bit means the successive element is greater than the current one and a 0 bit means the opposite. Thereafter, in the search phase, an algorithm is applied to filter the text T and find all the positions i in T such that $d_o(\phi(T_{i,m}), \phi(P)) \leq k$, where $T_{i,m} = t_i t_{i+1} \ldots t_{i+m-1}$ is the substring of T of length m starting at position i. The substrings $T_{i,m}$ are encoded into the binary string $\phi(T_{i,m})$ online in the same way as the pattern. The algorithm determines approximate matches of the transformed pattern $\phi(P)$ in the similarly transformed text $\phi(T)$. As these approximate matches are just the match candidates, they need to be verified using a

checking routine.

Verification. For verification, we use the reduction, by Gawrychowski and Uznanski, of the problem of k-isomorphism to the one of computing an heaviest increasing subsequence (Lemma 8, [23]). To compute the heaviest increasing subsequence, we use the algorithm of Jacobson and Vo [33], which runs in $O(m \log m)$ time for a sequence of length m. If we use a sorting algorithm with $O(m \log m)$ worst-case time complexity, the total time complexity of the verification is also $O(m \log m)$. In theory, the time complexity can be reduced to $O(m \log \log m)$ by using Han's sorting algorithm [27] and plugging a data structure which supports predecessor search in $O(\log \log m)$ time, such as van Emde Boas trees, in Jacobson and Vo's algorithm. Observe that in the simpler case where there are no repeated elements in u and v, deciding whether $u \approx_k v$ can be reduced to computing the longest increasing subsequence of $\pi(v)$, where π is a sorting permutation of u.

We propose two filtration algorithms, which are build on ideas from two algorithms for string matching with k mismatches, namely approximate SBNDM [28] and the GGF algorithm [25], respectively.

The first filtration algorithm, named AOPF1, is based on a generalization of the method used by approximate SBNDM and first proposed by Chang and Lawler [8] (lemma 3 described in Publication IV). Informally, the idea is to factorize a string x into substrings of another string y which cannot be extended to the right and are separated by 2-grams. It holds that the size r of this factorization satisfies $r - 1 \leq d_o(x, y)$.

Let $\hat{m} = |\phi(P)|$. The AOPF1 algorithm slides a window of size $\hat{m}$ along T, starting at position 0. For a given position i in T, the algorithm scans the substring $\phi(T_{i,m})$ from right to left and computes the factors F_j of $\phi(P)^r$ until either it has found $k + 2$ factors or it has scanned the whole substring. In the former case, by the lemma described above, the position is skipped. Otherwise the algorithm performs an additional filtration step, namely it computes $H(\psi(\pi(T_{i,m})), \psi(\pi(P)))$, where $\psi(u)$ is the string of length $|u| - 1$ such that

$$\psi(u)_i = \begin{cases} 1 & \text{if } u_i < u_{i+1} \\ 2 & \text{if } u_i = u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

and π is a sorting permutation of P, computed in the preprocessing phase.

The position is then verified only if $|H(\psi(\pi(T_{i,m})), \psi(\pi(P)))| \leq k$. Indeed, Lemma 2 in Publication IV can be easily proved to hold also when using $\psi(\pi(u))$ and $\psi(\pi(v))$ in place of $\phi(u)$ and $\phi(v)$ (observe that, if $u \approx_k v$, then $\pi(u) \approx_k \pi(v)$). We permute the strings with π so as to obtain a permutation of P where repeated elements are clustered, which allows us to perform a finer filtering using the ψ function. Note that, in principle, this additional filtration works with any permutation and ordering of repeated elements.

For example, if u is $(4, 1, 2, 4)$, v is $(4, 5, 2, 3)$ and π is the sorting permutation of u $2, 3, 1, 4$, we have $\pi(u) = (1, 2, 4, 4)$, $\pi(v) = (5, 2, 4, 3)$, $\psi(\pi(u)) = (1, 1, 2)$, $\psi(\pi(v)) = (0, 1, 0)$. Note that $d_o(\psi(\pi(u)), \psi(\pi(v))) = 2$, while $d_o(\phi(u), \phi(v)) = d_o(\psi(u), \psi(v)) = 1$, as $\phi(u) = \psi(u) = (0, 1, 1)$ and $\phi(v) = \psi(v) = (1, 0, 1)$.

The factors F_j are computed using the nondeterministic factor automaton of $\phi(P)^r$, which is simulated using a modified version of the bit-parallel SBNDM algorithm [43, 46]. The SBNDM algorithm is a slightly faster version of BNDM (Backward Nondeterministic DAWG Matching) [44] without bookkeeping of prefixes. The next scanned position is then $i + (\hat{m} - l) + 1$, where l is the length of the longest suffix of $\phi(T_{i,m})$ with at most $k + 1$ factors. The worst-case time complexity of this algorithm is $O(nm(\lceil m/w \rceil + \log m))$.

The second filtration algorithm, named AOPF2, is described informally as follows:

Let $H'(x, y)$ be the subset of the mismatch positions between x and y such that for each even position we exclude the two adjacent (odd) positions. Formally, we have

$$H'(x, y) = B_0 \cup B_1 \setminus (\{j - 1 : j \in B_0\} \cup \{j + 1 : j \in B_0\})$$

where B_0 ($B1$) is the set of the even (odd) mismatch positions, and it holds that $|H'(x, y)| \leq d_o(x, y)$ (see Lemma 4 in Publication IV).

For example, if $u = (4, 1, 2, 3)$ and $v = (4, 5, 3, 2)$ we have $u \approx_2 v$, $\phi(u) = (0, 1, 1)$, $\phi(v) = (1, 0, 0)$, $H(\phi(u), \phi(v)) = \{1, 3\}$, $H'(\phi(u), \phi(v)) = \{2\}$. In the preprocessing, the AOPF2 algorithm computes the bit-vector X of $\hat{m}$ bits such that the i-th bit is set to 1 if $P_i < P_{i+1}$ and to 0 otherwise. In other words X is the bit-vector encoding of $\phi(P)$. The algorithm then scans the text from left to right and maintains the bit-vector encoding Y of $\phi(T_{i,m})$, for $i = 1, \ldots, |T|$. For a given position i in T, the bit-vector

encodings of B_0 and B_1 are computed as $(X \wedge Y)$ & $01\ldots01$ and $(X \wedge Y)$ & $10\ldots10$, respectively. Then, we have that the bit-vector encoding of $H'(\phi(P), \phi(T_{i,m}))$ is equal to

$$B_0 \mid B_1 \mathbin{\&} {\sim}((B_0 \ll 1) \mid (B_0 \gg 1))\,.$$

The size of $H'(\phi(P), \phi(T_{i,m}))$ is computed using the sideways addition operation SA on each word of the resulting bit-vector. Given a word X, the sideways addition of X returns the number of bits set in X. This operation can be computed in $O(\log\log w)$ time in the word-RAM model [49] and is also available as a POPCNT instruction in recent processors of the x86 family. The worst-case time complexity of this algorithm is $O(n(\lceil m/w\rceil \log\log w + m\log m))$. The space complexity of both algorithms is $O(\lceil m/w\rceil)$. The pseudocode of the two algorithms is shown in Alg. 8 and 9. The **psi-filter** procedure called in AOPF1 at line 18 performs the additional filtration step based on the ψ function and calls the verification procedure, if necessary.

7.3 Analysis

In this section we analyze the average-case running time of the AOPF1 algorithm, and show that it is sublinear on average if k is not too large. Suppose that T is a uniformly random string over an alphabet Σ of size σ. The string $\phi(T)$ is not uniformly random in general as $Pr[\phi(T)_i = 1] = (\sigma+1)/(2\sigma)$ and $Pr[\phi(T)_i = 0] = (\sigma-1)/(2\sigma)$. We make the simplifying assumption that either all the symbols of T are distinct, in which case the distribution becomes uniform, or that the alphabet is large enough so that the distribution is arbitrarily close to uniform. Assume that $k < m/(\log_\sigma m + O(1))$ and let X_j be the random variable corresponding to the length of factor F_j. By the "Main Lemma" of Chang and Lawler [8] we obtain that

1. the probability $Pr[X_1 + X_2 + \ldots + X_{k+1} \geq m]$ of a verification using lemma 3 of Publication IV is less than $1/m^3$;

2. $E[X_j] < \log_\sigma m + 3$;

since skipping two symbols instead of one between each factor F_j does not invalidate the assumption that the variables X_j are independent and identically distributed. By (1), the total verification time is thus

Algorithm 8 AOPF1(P, T, k)

```
 1: m̂ ← |P| − 1
 2: B[0] ← B[1] ← 0^m̂
 3: E ← 1^m̂
 4: for i ← 1 to m̂ do
 5:    c ← 0
 6:    if P_i < P_{i+1} then
 7:       c ← 1
 8:    end if
 9:    B[c] ← B[c] | (1 ≪ (i − 1))
10: end for
11: i ← m̂ + 1
12: while i ≤ |T| do
13:    (e, j, D) ← (0, 0, E)
14:    while e ≤ k and j < m̂ do
15:       j ← j + 1
16:       c ← 0
17:       if T_{i−j} < T_{i−j+1} then
18:          c ← 1
19:       end if
20:       D ← (D ≫ 1) & B[c]
21:       if D = 0^m̂ then
22:          (e, j, D) ← (e + 1, j + 1, E)
23:       end if
24:    end while
25:    if j ≥ m̂ and e ≤ k then
26:       psi-filter(P, T, i)
27:    end if
28:    i ← i + (m̂ − min(j, m̂)) + 1
29: end while
```

Algorithm 9 AOPF2(P, T, k)

```
1:  m̂ ← |P| − 1
2:  X ← Y ← 0^m̂
3:  C[0] ← C[1] ← 0^m̂
4:  for i ← 1 to m̂ do
5:      j ← i mod 2
6:      C[j] ← C[j] | (1 ≪ (i − 1))
7:      if P_i < P_{i+1} then
8:          X ← X | (1 ≪ (i − 1))
9:      end if
10:     if T_i < T_{i+1} then
11:         Y ← Y | (1 ≪ (i − 1))
12:     end if
13: end for
14: for i ← m̂ to |T| − 1 do
15:     if T_i < T_{i+1} then
16:         Y ← Y | (1 ≪ (i − 1))
17:     end if
18:     B_0 ← (X ^ Y) & C[0]
19:     B_1 ← (X ^ Y) & C[1]
20:     W ← (B_0 ≪ 1) | (B_0 ≫ 1)
21:     e ← SA(B_0 | B_1 & ∼W)
22:     if e ≤ k then
23:         verify (P, T, i)
24:     end if
25:     Y ← Y ≫ 1
26: end for
```

$O((n/m^3)m \log m)$. Instead, by (2), it follows that the average number of symbols scanned in a single window and the average shift length are equal to $(k+1)(\log_\sigma m+3)$ and $m-(k+1)(\log_\sigma m+3)$, respectively. From this we obtain that the average filtering time is $O((n/m)k \log_\sigma m)$ for the aforementioned choice of k. Hence, the running time of both phases is sublinear on average.

7.4 Experiments

We tested AOPF1 and AOPF2 against the following algorithms:

- AOPF1b: the filtration method based on the Hamming distance using Approximate SBNDM;
- AOPF2b: the filtration method based on the Hamming distance using the GGF algorithm;
- naive: the naive method where all the text positions are checked.

Note that the AOPF1b and AOPF2b algorithms must use $2k$ as bound on the number of mismatches. In the AOPF1b algorithm we employ the same additional filtration step used in AOPF1.

Table 7.1 shows the average execution times of the algorithms for the humidity data in 10 of milliseconds for $k \in \{1,2,3\}$. In addition, a graph of the times for the data and $k=1$ (with logarithmic scale on the y axis) is shown in Fig. 7.1. All the algorithms use the verification method described in Sect. 7.3.

From the results, we observe that i) AOPF1 and AOPF2 are significantly faster than the naive method, except for the case when $m=5$ and $k=2$ and 3; ii) AOPF1 is always faster than AOPF1b; iii) AOPF2 is either faster or comparable to AOPF2b. For all tested values of k and m, in most cases AOPF1 is the fastest algorithm. The results vary slightly from the results in Publication IV as AOPF2 is slower for relative humidity data as compared to Dow Jones and Helsinki temperature data.

Table 7.1. Execution times of the algorithms (in 10 of milliseconds) for relative humidity data.

Relative humidity					
$k = 1$					
m	AOPF1	AOPF1b	AOPF2	AOPF2b	naive
5	45.2	45.9	37.1	43.2	53.3
10	8.1	17.1	21.7	35.1	209.9
15	4.0	10.6	12.8	20.7	395.8
20	2.4	6.1	10.1	14.9	607.9
25	1.9	4.0	7.9	10.6	855.4
30	1.6	2.8	7.0	7.7	1144.7
50	0.7	1.2	6.1	5.8	3243.8
$k = 2$					
m	AOPF1	AOPF1b	AOPF2	AOPF2b	naive
5	87.1	63.9	59.7	58.6	54.4
10	54.40	55.9	79.6	117.4	207.9
15	26.2	37.3	33.4	71.8	398.2
20	9.7	36.2	21.1	46.9	616.8
25	5.6	25.9	14.2	33.2	851.1
30	3.6	16.3	9.8	19.8	1106.2
50	1.7	3.9	6.8	7.9	2414.2
$k = 3$					
m	AOPF1	AOPF1b	AOPF2	AOPF2b	naive
5	63.8	64.6	59.3	58.4	53.2
10	147.1	150.6	170.0	197.3	209.7
15	45.4	46.7	98.7	185.8	395.0
20	33.8	53.2	49.2	116.6	608.8
25	19.7	57.3	31.0	84.8	849.7
30	11.0	52.8	18.3	55.2	1103.4
50	2.9	15.6	7.9	14.7	2462.7

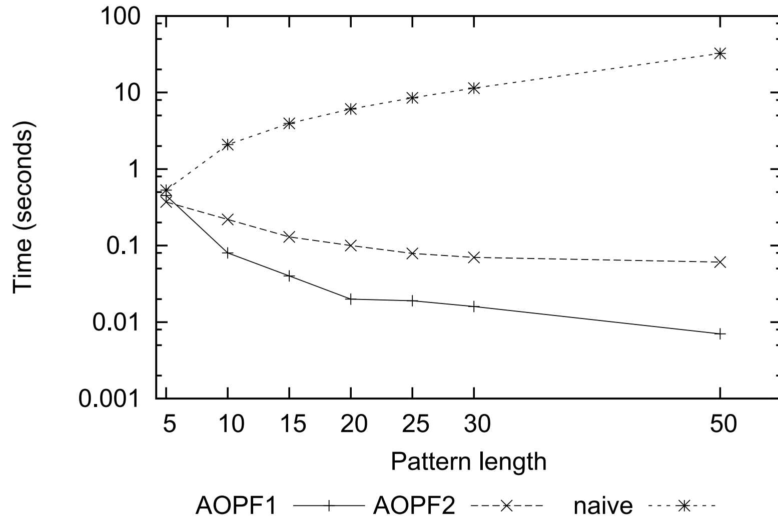


Figure 7.1. Execution times of algorithms for humidity data

8. Conclusions

We present several practical algorithms for the order-preserving matching problem and its approximate variant in this thesis. We have proved with our experimental tests that our algorithms are effective and faster than the previous solutions in most cases. In the publications we have used the Dow Jones index, feature data, Helsinki temperature data and random data for testing. We chose a different data set for testing in our thesis, relative humidity data. The tests were performed with a wide range of pattern lengths.

Since not many efficient algorithms had been proposed for the order-preserving matching problem, we first introduced a simple algorithm based on filtration (Publication I), where in the non-matching positions in the text are filtered out. Any exact string matching algorithm can be used as the filtration algorithm. We used SBNDM2 and SBNDM4 [15], FSO [22] and KMP [38] as the filtration algorithms. We carried out tests and found that our solutions based on SBNDM2 and SBNDM4 were faster than the previous solutions in most cases and the FSO based solution was faster for short patterns.

Later, we combined the SIMD instruction set architecture with filtration (Publication II). The SIMD architecture requires careful redesigning of an algorithm, and the outcome is not necessarily efficient for an arbitrary string matching problem. However, we succeeded in developing two online solutions which were faster than our previous algorithm (in Publication I). The two online solutions used the SSE and AVX instruction set architecture. SIMD instructions were originally developed for multimedia but are recently employed for pattern matching. Our results show that SIMD instructions can also be very efficient in order-preserving matching as well.

We also developed an offline solution based on the FM-index (Publica-

tion II) and it is superior for long patterns. However, the search algorithm of the offline solution was slower than we expected for short patterns because of cache inefficiency. Another possible reason for the slowness of FM-index is probably that locate is a slow operation in FM-index.

It was thought that there might be inefficiency in the FM-index for a bit string. It is because the FM-index uses a wavelet tree, and it would be useless in the case of a binary text. So a modified FM-index without a wavelet tree might be more efficient. Therefore we implemented another FM-index without a wavelet tree. To keep the FM-index compressed, the Burrows-Wheeler transform of the bit-string was computed and was compressed via rank and select dictionaries, and then the backward search on the compressed bit string was implemented via rank/select queries. However, we observed that this approach was slower than the standard one.

Henceforth, we presented another practical solution without filtration for the order-preserving matching problem. Again we employed the SSE instruction set architecture. Our results in the Publication III show that our solution is the fastest until m is 20. However, the results differ a bit in the case of relative humidity data, in which the solution is the fastest for all values of m except when m reaches 50.

We also provided practical solutions for approximate order-preserving matching grounded on filtration. And one of the solutions is the first sublinear solution for the problem. We compared our solutions against the naive solution since no practical solution is available to date for the approximate variant of the problem. Our solutions were faster than the naive solution in most cases.

It seems to be feasible that many more effective solutions can be developed using the SIMD instruction set architecture for order-preserving matching and its variant. Moreover, the solutions of order-preserving matching and its variant can be extended to the multi-pattern [50] and multi-dimensional case. Such methods in turn can be applied to electronic medical record (EMR) to find useful patterns for detecting adverse medical conditions.

Bibliography

[1] J. Ardnt. Matters computational. http://www.jjj.de/fxt/ (Loaded in Jan. 2016).

[2] Ricardo A. Baeza-Yates and Mireille Régnier. Average running time of the Boyer–Moore–Horspool algorithm. *Theor. Comput. Sci.*, 92(1):19–31, 1992.

[3] Belazzougui, Adeline Pierrot, Mathieu Raffinot, and Stéphane Vialette. Single and multiple consecutive permutation motif search. In *Algorithms and Computation - 24th International Symposium, ISAAC 2013, Hong Kong, China, December 16–18, 2013, Proceedings*, pages 66–77, 2013.

[4] Oren Ben–Kiki, Philip Bille, Dany Breslauer, Leszek Gąsieniec, Roberto Grossi, and Oren Weimann. Optimal packed string matching. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12–14, 2011, Mumbai, India*, pages 423–432, 2011.

[5] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[6] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, DEC SRC, 1994.

[7] Domenico Cantone, Simone Faro, and M. Oguzhan Külekci. An efficient skip–search approach to the order-preserving pattern matching problem. In *Proceedings of the Prague Stringology Conference 2015, Prague, Czech Republic, August 24–26, 2015*, pages 22–35, 2015.

[8] William I. Chang and Eugene L. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, 1994.

[9] Christian Charras, Thierry Lecroq, and Joseph Daniel Pehoushek. A very fast string matching algorithm for small alphabeths and long patterns (extended abstract). In *Combinatorial Pattern Matching, 9th Annual Symposium, CPM 98, Piscataway, New Jersey, USA, July 20–22, 1998, Proceedings*, volume 1448 of *Lecture Notes in Computer Science*, pages 55–64. Springer, 1998.

[10] Yanping Chen, Eamonn Keogh, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, and Gustavo Batista. The UCR time series classification archive. https://www.cs.ucr.edu/∼eamonn/UCRsuite.html (Loaded in Jan. 2016).

[11] Tamanna Chhabra and Jorma Tarhio. Order-preserving matching with filtration. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 – July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 307–314. Springer, 2014.

[12] Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. Fast order-preserving pattern matching. In *Combinatorial Optimization and Applications - 7th International Conference, COCOA 2013, Chengdu, China, December 12–14, 2013, Proceedings*, volume 8287 of *Lecture Notes in Computer Science*, pages 295–305. Springer, 2013.

[13] Sukhyeun Cho, Joong Chae Na, Kunsoo Park, and Jeong Seop Sim. A fast algorithm for order-preserving pattern matching. *Inf. Process. Lett.*, 115(2):397–402, 2015.

[14] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Marcin Kubica, Alessio Langiu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Order-preserving incomplete suffix trees and order-preserving indexes. In *String Processing and Information Retrieval -20th International Symposium, SPIRE 2013, Jerusalem, Israel, October 7–9, 2013, Proceedings*, volume 8214 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2013.

[15] Branislav Durian, Jan Holub, Hannu Peltola, and Jorma Tarhio. Improving practical exact string matching. *Inf. Process. Lett.*, 110(4):148–152, 2010.

[16] Simone Faro and M. Oguzhan Külekci. Fast multiple string matching using streaming SIMD extensions technology. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21–25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 2012.

[17] Simone Faro and M. Oguzhan Külekci. Fast packed string matching for short patterns. In *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013*, pages 113–121, 2013.

[18] Simone Faro and M. Oguzhan Külekci. Fast and flexible packed string matching. *J. Discrete Algorithms*, 28:61–72, 2014.

[19] Simone Faro and M. Oguzhan Külekci. Efficient algorithms for the order preserving pattern matching problem. *CoRR*, abs/1501.04001, 2015.

[20] Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2):13, 2013.

[21] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12–14 November 2000, Redondo Beach, California, USA*, pages 390–398, 2000.

[22] Kimmo Fredriksson and Szymon Grabowski. Practical and optimal string matching. In *String Processing and Information Retrieval, 12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2–4, 2005, Proceedings*, volume 3772 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2005.

[23] Pawel Gawrychowski and Przemyslaw Uznanski. Order-preserving pattern matching with k mismatches. *CoRR*, abs/1309.6453, 2013.

[24] Pawel Gawrychowski and Przemyslaw Uznanski. Order-preserving pattern matching with k mismatches. In *Combinatorial Pattern Matching - 25th Annual Symposium, CPM 2014, Moscow, Russia, June 16–18, 2014. Proceedings*, pages 130–139, 2014.

[25] Emanuele Giaquinta, Szymon Grabowski, and Kimmo Fredriksson. Approximate pattern matching with k-mismatches in packed text. *Inf. Process. Lett.*, 113(19–21):693–697, 2013.

[26] Simon Gog. Succinct data structure library 2.0. `https://github.com/simongog/sdsl-lite` (Loaded in Jan. 2016).

[27] Yijie Han. Deterministic sorting in O($n$loglog$n$) time and linear space. *J. Algorithms*, 50(1):96–105, 2004.

[28] Tommi Hirvola and Jorma Tarhio. Approximate online matching of circular strings. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 – July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 315–325. Springer, 2014.

[29] R. Nigel Horspool. Practical fast searching in strings. *Softw., Pract. Exper.*, 10(6):501–506, 1980.

[30] Andrew Hume and Daniel Sunday. Fast string searching. *Softw., Pract. Exper.*, 21(11):1221–1248, 1991.

[31] Intel. Intel Architecture Instruction Set Extensions Programming Reference. `https://software.intel.com/sites/default/files/m/9/2/3/41604` (Loaded in Jan. 2016).

[32] Intel. Intel (R) 64 and IA-32 Architectures Software Developer's Manual. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html` (Loaded in Jan. 2016).

[33] Guy Jacobson and Kiem–Phong Vo. Heaviest increasing/common subsequence problems. In *Combinatorial Pattern Matching, Third Annual Symposium, CPM 92, Tucson, Arizona, USA, April 29 – May 1, 1992, Proceedings*, volume 644 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1992.

[34] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, and Seok–Ho Myung. Performance of SSE and AVX instruction sets. *CoRR*, abs/1211.0820, 2012.

[35] James H. Morris Jr. and Vaughan R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

[36] Juha Kärkkäinen. Personal communication, 2016.

[37] Jinil Kim, Peter Eades, Rudolf Fleischer, Seok–Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order- preserving matching. *Theor. Comput. Sci.*, 525:68–79, 2014.

[38] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[39] Marcin Kubica, Tomasz Kulczynski, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. A linear time algorithm for consecutive permutation pattern matching. *Inf. Process. Lett.*, 113(12):430–433, 2013.

[40] M. Oguzhan Külekci. Filter based fast matching of long patterns by using SIMD instructions. In *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 – September 2, 2009*, pages 118–128, 2009.

[41] Susana Ladra, Oscar Pedreira, José Duato, and Nieves R. Brisaboa. Exploiting SIMD instructions in current processors to improve classical string algorithms. In *Advances in Databases and Information Systems - 16th East European Conference, ADBIS 2012, Poznań, Poland, September 18–21, 2012. Proceedings*, pages 254–267, 2012.

[42] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22–24 January 1990, San Francisco, California*, pages 319–327, 1990.

[43] Gonzalo Navarro. NR-grep: A fast and flexible pattern-matching tool. *Softw., Pract. Exper.*, 31(13):1265–1312, 2001.

[44] Gonzalo Navarro and Mathieu Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, 5:4, 2000.

[45] Gonzalo Navarro and Mathieu Raffinot. Flexible pattern matching in strings - practical on-line search algorithms for texts and biological sequences. 2002.

[46] Hannu Peltola and Jorma Tarhio. Alternative algorithms for bit-parallel string matching. In *String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 8–10, 2003, Proceedings*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2003.

[47] Peter Sanders. Algorithm engineering - an attempt at a definition using sorting as an example. In *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*, pages 55–61, 2010.

[48] Jorma Tarhio and Hannu Peltola. String matching in the DNA alphabet. *Softw., Pract. Exper.*, 27(7):851–861, 1997.

[49] Sebastiano Vigna. Broadword implementation of rank/select queries. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30–June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2008.

[50] Bruce Watson. Personal communication, 2015.

Errata

Corrections to Publication II

- The analysis of online solutions is not correct. Look at the analysis in Sect. 5.1 in the summary.

Corrections to Publication III

- Page 5,

  wspc(A, B)
  $B \leftarrow$ _mm_cmpgt_epi8(B, A)
  $r \leftarrow$ _mm_movemask_epi8(B)
  return r

- Page 5, line 3: Replace $A[j] \leq B[j]$ with $A[j] < B[j]$

- Page 6, Fig. 2, line 10: if $C = 0$ then goto out

- Page 6, before Example 2: Replace "When $m = O(\alpha)$" by "In the average case".

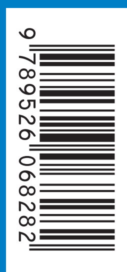
- Page 6, line 11: Replace $A[j] \leq B[j]$ with $A[j] < B[j]$ and $y[i+h+r(j)] \leq y[i+h+r(j+1)])$ with $y[i+h+r(j)] < y[i+h+r(j+1)])$

- Page 6, line 14: Replace $y[i+h+r(j)] \leq y[i+h+r(j+1)])$ with $y[i+h+r(j)] < y[i+h+r(j+1)])$

Aalto-DD 101/2016

String matching is a widely studied problem in Computer Science. There have been many recent developments in this field. One fascinating problem considered lately is the order-preserving matching (OPM) problem. The task is to find all the substrings in the text which have the same length and relative order as the pattern, where the relative order is the numerical order of the numbers in a string. The problem finds its applications in the areas involving time series or series of numbers. More specifically, it is useful for those who are interested in the relative order of the pattern and not in the pattern itself. For example, it can be used by analysts in a stock market to study movements of prices.

We proposed various sublinear solutions for exact and approximate OPM and we show with experimental tests that our solutions are efficient than the previous solutions.



ISBN 978-952-60-6828-2 (printed)
ISBN 978-952-60-6829-9 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

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

BUSINESS +
ECONOMY

ART +
DESIGN +
ARCHITECTURE

SCIENCE +
TECHNOLOGY

CROSSOVER

DOCTORAL
DISSERTATIONS