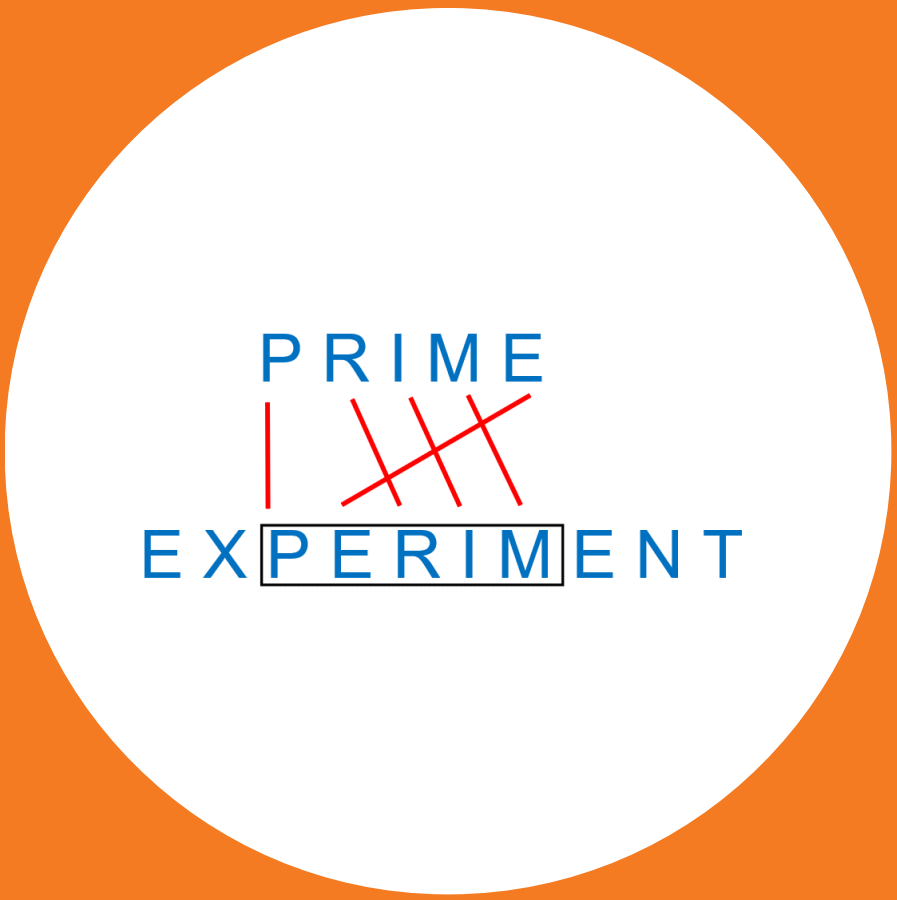
Department of Computer Science

Improved Online Algorithms for Jumbled Matching

Sukhpal Singh Ghuman



Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 242/2017

Improved Online Algorithms for Jumbled Matching

Sukhpal Singh Ghuman

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 15 January 2018 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 Bruce Watson, Stellenbosch University, South Africa
Adjunct Professor Kimmo Fredriksson, University of Eastern Finland, Finland

Opponent
Associate Professor M. Oğuzhan Külekci, Istanbul Technical University, Turkey

Aalto University publication series
DOCTORAL DISSERTATIONS 242/2017

ISBN 978-952-60-7757-4 (printed)
ISBN 978-952-60-7758-1 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-7758-1

Unigrafia Oy
Helsinki 2017

Finland

441 697
Printed matter

Aalto University

Abstract

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

Author
Sukhpal Singh Ghuman

Name of the doctoral dissertation
Improved Online Algorithms for Jumbled Matching

Publisher School of Science

Unit Department of Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 242/2017

Field of research Software Technology

Manuscript submitted 21 August 2017 **Date of the defence** 15 January 2018

Permission to publish granted (date) 17 October 2017 **Language** English

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

Abstract

In Computer Science, the problem of finding the occurrences of a given string is a common task. There are many different variations of the problem. We consider the problem of jumbled pattern matching (JPM) (also known as Abelian pattern matching or permutation matching) where the objective is to find all permuted occurrences of a pattern in a text. Jumbled pattern matching has numerous applications in the field of bioinformatics. For instance, jumbled matching can be used to find those genes that are closely related to one another. Besides exact jumbled matching we study approximate jumbled matching where each occurrence is allowed to contain at most k wrong or superfluous characters. We present online algorithms applying bitparallelism to both types of jumbled matching. Two of our algorithms are filtration methods applying SIMD (Single Instruction Multiple Data) computation. Furthermore, we have developed a bitparallel algorithm for episode matching. This algorithm finds the maximal parallel episodes of a given sequence. Most of the other new algorithms are variations of earlier methods. We show by practical experiments that our algorithms are competitive with previous solutions.

Keywords

ISBN (printed) 978-952-60-7757-4 **ISBN (pdf)** 978-952-60-7758-1

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

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2017

Pages 118 **urn** http://urn.fi/URN:ISBN:978-952-60-7758-1

Preface

First of all, I would like to thank my supervisor, Professor Jorma Tarhio for guiding me throughout my research and giving me his valuable time whenever I needed. He introduced me to various interesting problems in Stringology. I also thank God for his grace.

I would like to thank Hannu Peltola for his help and advice. I express my gratitude to Tommi Hirvola and Simone Faro for sharing their ideas. I was privileged to work with colleagues like Emanuele Giaquinta, Kerttu Pollari-Malmi and Tamanna. I wish to thank them all.

I am thankful to my pre-examiners Dr. Kimmo Fredriksson and Prof. Bruce Watson for reviewing my dissertation and for their valuable remarks. I also appreciate Prof. Sami Khuri and Prof. M. Oğuzhan Külekci, for their comments and suggestions.

I am highly grateful to my uncle, Prof. Ranjit Singh Ghuman for his guidance and encouragement. I also want to thank my sister Sukhwinder Kaur for her motivation and support. I acknowledge my wife Tamanna for bearing me and helping me throughout. Last but not least, I am very thankful to my mother Gurbax Kaur for encouraging and supporting me.

Mississauga, Canada

November 2017

Sukhpal Singh Ghuman

Contents

1. Introduction

The problem to find the occurrences of a given string is a common task that has gained much attention due to the enormous amount of information that is available these days. Also, extracting relevant information is an even more important goal. The problem of string matching [53] consists of two strings, a text and a pattern, and the task is to find all occurrences of the pattern in the text.

In the recent past, numerous variations of this problem have appeared, such as exact string matching, approximate string matching, circular sting matching [63], order preserving matching [2, 17, 43], jumbled pattern matching [9, 14] and many more. The task of exact string matching is to find all occurrences of the pattern P in the text T, i.e. all positions i such that $t_i t_{i+1} \cdots t_{i+m-1} = p_1 p_2 \cdots p_m$ where t_i and p_j represent single characters in T and P respectively. In approximate string matching, the objective is to find the substrings of T whose edit distance to P is at most k or which have at most k mismatches with P, where $0 \leq k < m$.

In this thesis, we address the problem of jumbled pattern matching (JPM) [9, 14] that is an interesting variation of string matching. The task is to find the substrings of T that are permutations of P. For instance, a permutation of abcb occurs in cdfbacbda. This is an example of exact jumbled matching. We consider also approximate jumbled matching. The main focus of this thesis is to improve the practical speed of jumbled pattern matching.

1.1 Algorithm Engineering

Algorithms and data structures are the basic building blocks of every computer application. Algorithm engineering as defined by Sanders [58], is a cycle of design, analysis, implementation and experimentation that leads

to new design ideas. As experiments play an important role in developing algorithms, algorithm engineering [16] over the past few years has become a well known definition for experimental evaluation in the context of algorithm development. In addition to that, the goal of algorithm engineering is also to speed up the process of transferring algorithmic knowledge into applications.

In other words, algorithm engineering is a process that is based on a full feedback loop. It begins with the design of an algorithm, followed by the analysis, implementation and experimental evaluation. The results of the experimental evaluation can then be reused for modifications and improvement of the algorithm. Hence, we present our algorithms with practical experiments to determine whether our new solutions are better than the previous solutions.

1.2 Research goal

The research goal of this thesis is to develop more efficient algorithms for exact and approximate jumbled pattern matching and to compare our new solutions with the earlier solutions.

1.3 Main results

We have developed online algorithms for exact and approximate jumbled matching. Online algorithms can preprocess the pattern, but they do not preprocess the text. We present 16 new algorithms. Most of our algorithms apply bitparallelism and some of them use novel techniques in this context, such as SIMD (Single Instruction Multiple Data) [37] and shared counters. We carried out extensive practical experiments to compare the new algorithms with the best earlier algorithms. The new algorithms are competitive in most scenarios.

BAM2, EBL and EFB are our best algorithms for exact jumbled matching. BAM2 is a modification of an algorithm by Cantone and Faro [11]. For the texts of large alphabets (English and proteins), BAM2 outperforms all algorithms and is at least two times faster than the older methods. EBL is based on an exact string matching algorithm by Ďurian et al. [19]. In the case of DNA data, EBL is the most efficient. Both EBL and BAM2 show sublinear behavior in the experiments. EFB is a modification of an algo-

rithm by Navarro [52]. In the case of binary data, EFB outperforms all other algorithms. For approximate jumbled matching, we introduce four competitive algorithms.

In addition to that, we introduce two improved algorithms for exact jumbled matching for short patterns using SIMD computation. SIMD is used to filter out the text and then apply a standard method of JPM as a subroutine. Our experiments show that the best algorithm is 30% faster than previous algorithms for short English patterns.

Furthermore, we have developed a bitparallel algorithm for episode matching [47] that finds the maximal parallel episodes from the given sequence.

1.4 Structure of the thesis

The remaining of the thesis is organized as follows. Chapter 2 contains the common definitions and notations used throughout the thesis and reviews the background that is important and necessary to introduce the approaches and results described in the following chapters.

Chapter 3 presents the formal definition of exact and approximate JPM. In addition to that, we discuss the earlier cost models for JPM. We also shine light on the applications of JPM problems. Chapter 4 describes the previous methods for JPM and its variations.

Chapter 5 focuses on exact JPM, while presenting new algorithms for exact jumbled matching. Chapter 6 introduces our methods that apply SIMD computation in order to quickly filter the text for exact JPM. Chapter 7 elaborates on our new solutions for approximate jumbled matching. Chapter 8 describes episode matching with JPM. Chapter 9 presents the experimental results and Chapter 10 concludes the thesis and discusses the future scope of the problem.

1.5 Contribution of the Author

This thesis includes material from the original publications [15, 27]. The co-authors made a contribution to some of the results. This thesis also contains several new algorithms that have not been presented before. Most of these algorithms are joint work with Jorma Tarhio. Chapter 7 contains two unpublished algorithms suggested by Simone Faro and Tommi Hir-

vola. The author implemented all the algorithms and performed all the experiments.

2. Background

2.1 Definitions and Notations

A non-empty set of characters is called an alphabet. We denote an alphabet by the symbol Σ and the size of an alphabet by $|\Sigma|$. An alphabet may be an English alphabet, an integer alphabet and so on. We mainly consider English data, DNA data, protein data and binary data for testing our solutions. The symbol w indicates the number of bits in a computer word. Unless otherwise stated, log means log_2.

A string is a finite sequence of characters over an alphabet Σ. A string S is represented as $S = s_0 s_1 \cdots s_{n-1}$, where n is the length of the string. The length of a string S is denoted by $|S|$. Let {A,B,C} be the alphabet Σ, then ABACABAC is a string of length $|S|$ = 8 over the alphabet. Additionally, a subsequence of a string S is a string S' such that S' can be obtained from S by deleting some characters of S, without changing the order of the remaining characters. A substring of a string $S = s_0 s_1 \cdots s_{n-1}$ is $S'' = s_i \cdots s_j$ where $0 \leq i, j < n$. A prefix of a 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 $i \geq 0$. The concatenation of two strings S and T is denoted by $S.T$.

The symbol s_i or $s[i]$ denotes the i^{th} symbol of S, for $0 \leq i < |S|$ and $S^r = s_{n-1} s_{n-2} \ldots s_0$ is the reverse of the string S. The edit distance between two strings is the minimum number of edit operations that can transform one string to another. The total number of edit operations corresponds to the total number of insertions, deletions or substitutions of characters in a string. In a case when only the edit operation substitution is allowed, the edit distance is called the Hamming distance. For example, if $S = abaabba$ and $T = abbabbb$, then the Hamming distance between S and T is 2.

2.2 String Matching

String matching [53] is a common problem in computer science. Let $T = t_0 t_1 \cdots t_{n-1}$ and $P = p_0 p_1 \cdots p_{m-1}$ be the text and pattern with length m and n respectively, over a finite alphabet Σ. The task of exact string matching is to find all occurrences of P in T, i.e. to find all positions i such that $t_i t_{i+1} \cdots t_{i+m-1} = p_0 p_1 \cdots p_{m-1}$.

2.2.1 Approximate String Matching

A variation of the string matching problem can be obtained by allowing the matches to be approximate, so as to search for the substrings of the text T that are *similar* to the pattern P. It is also known as the k mismatches problem, where the task is to find the substrings of T that are at Hamming distance at most k from P, i.e., a substring of T matches P with at most k mismatches.

2.2.2 Bitvectors

A bitvector is denoted by $B = b_0 b_1 \cdots b_{n-1}$, where b_i is 0 or 1. Two important bits in a bitvector are the least significant bit b_{n-1} and the most significant bit b_0. Numerous bit operators can be used in the bitvector operations. The operator | denotes the bitwise OR operator, $\&$ depicts the bitwise AND operator and $\sim$ represents the negation operator. Another operator $<<$ is the left shift operator which shifts the specified bits to the left and inserts zeros starting from the least significant bit. Similarly, the $>>$ operator shifts the bits right by inserting zeros into the most significant bit. Remaining arithmetic operations such as addition and subtraction are defined as normal binary operations.

2.2.3 Bitparallelism

Bitparallelism allows the execution of various operations simultaneously over the bits of a single computer word of length w. With a single operation, the bits can be updated using the internal parallelism of bit operations. Due to parallel operations, the number of operations are reduced by a factor of up to w, thus significant amount of speedup can be achieved. Several faster string algorithms have been introduced using this technique.

2.2.4 q-grams

A substring of q characters of a string $S = s_0 s_1 \cdots s_{n-1}$ is called a q-gram. For example, let $S = abbababbcbacc$, then substring $u = abab$ is a 4-gram. Processing a q-gram instead of a single character often helps to increase the performance of an algorithm. As in the SBNDM2 algorithm [19], 2-gram methodology is applied and has proved to be effective.

2.2.5 SBNDM

SBNDM [56] (Simplified Backward Nondeterministic DAWG Matching) for exact string matching is a variation of BNDM (Backward Nondeterministic DAWG Matching) [50, 51]. SBNDM2 [19] is a 2-gram version of SBNDM. In SBNDM2, two characters are read, before the state vector is tested and two characters are matched at the end of the pattern. It works on bitparallelism by recognizing the factors of the pattern by simulating a non-deterministic automaton of the reversed pattern. A state vector is maintained such that there is a one in every position where a factor of the pattern begins. The factor is a suffix of the processed string of the text. For instance, consider a text substring $CAGT$ and let the pattern be $CAGTATGCAGT$, then the value of the state vector is 10000001000.

2.3 SIMD

SIMD (Single Instruction Multiple Data) [37] is a type of parallel architecture that allows one instruction to operate at the same time on multiple data items. SIMD was initially used in multimedia especially in processing images or audio files, and found applications in many areas, such as multimedia and cryptography. Recently, SIMD architecture has also been used in string matching [45, 46].

A detailed review of SIMD and its applications can be found in [32]. In the following, we consider SIMD computation only in the Intel/AMD x64_86 architecture. SSE (Streaming SIMD Extensions) [40] comprise SIMD instruction sets supported by modern processors. SIMD instructions use sixteen 128-bit registers known as XMM0, XMM1,$\cdots$, XMM15. Modern microprocessors support the parallel execution of operations on multiple data simultaneously through a set of special instructions that work on the limited number of special registers. For example, one can

multiply or add several numbers simultaneously in parallel.

The SIMD instruction set architecture [40] is based on the word RAM model which assists in speeding up the performance of string matching algorithms. The computer carries out operations on computer words and blocks of characters. Numerous operations on items occupying a single word are assumed to be achieved in constant time, that results in better performance of string matching algorithms. SIMD programming can be implemented using intrinsic functions. An intrinsic function is a function whose implementation is handled especially by the compiler.

The SIMD instruction set supports both packed and scalar string matching instructions. Scalar operations apply an operation on a single value whereas in *packed string matching* [23, 24], sets of adjacent characters are packed into one single word, according to the size of the word in the target machine. Various packed instructions based on the SIMD technology are used to design efficient solutions for jumbled matching. This allows us to compare sets of the characters in bulk, rather than individually, by comparing the corresponding words. In this case, the symbol w indicates the length of the SIMD register (= 128).

The 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 (discussed in forthcoming paragraphs) 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 different intrinsic functions that 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 different versions of SIMD extensions like MMX, SSE, SSE2, SSE3, SSE4, SSE4.1, SSE4.2, AVX and so on. We explain MMX, SSE and AVX as follows:

- *MMX (MultiMedia eXtention):* MMX was introduced in the Pentium processor by Intel. It uses 64-bit registers and can handle only integer data. Therefore its use has seen a decline in recent years.

- *SSE (Streaming SIMD Extensions):* Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in the Pentium III processors. It uses a 128-bit register and can therefore handle single precision floating point numbers. SSE2 (Streaming SIMD Extensions 2) was first introduced by Intel with the initial version of the Pentium 4 in 2001. It extends the earlier SSE instruction set, and was intended to fully replace MMX. SSE2 extended the functionality of SSE and can process two double precision floating point numbers simultaneously, thereby providing important speedups in algorithms. Intel extended SSE2 to create SSE3 in 2004. Thereafter, SSE4 was introduced on September 2006. SSE4 introduced 54 new instructions, of which 47 instructions comprises SSE4.1 and the remaining 7 instructions referred to as SSE4.2. String manulpation operators (such as cmpestr) were introduced in SSE4.2.

  SSE has 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

  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.

- *AVX (Advanced Vector eXtensions):* The AVX instruction set was introduced 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. It also introduced three operand instructions. AVX2 contains the extension of AVX, by using 256 bit numeric processing capabilities instructions instead of 128 bit SIMD integers instructions. There were some memory alignment issues that were present in SSE. These issues were relaxed in AVX2.

We have practiced some intrinsic functions to carry out the implementation of our proposed algorithms. The intrinsic functions used are:
1. Comparison Function (_mm_cmpeq_epi8): The compiler intrinsic equivalent of the function is

$$\_\_m128i \ \_mm\_cmpeq\_epi8\ (\_m128i\ x,\ \_m128i\ y).$$

It compares packed 8-bit integers in x and y bytewise for equality and stores the result. 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 all 1's, otherwise, it is set to 0.

2. Mask Function (_mm_movemask_epi8): The compiler intrinsic equivalent of the function is

$$\_mm\_movemask\_epi8\ (\_\_m128i\ \_mm\_cmpeq\_epi8\ (\_\_m128i\ x,\ \_\_m128i\ y)).$$

It creates a mask from the most significant bit of each 8-bit element in the parameter z, and stores the result.

3. Load function (_mm_load_si128): The compiler intrinsic equivalent of the function is

$$\_\_m128i \ \_mm\_loadu\_si128\ (\_\_m128i\ const* mem\_addr).$$

This load instruction loads 128 bits of integer data from the memory location.

4. Extract Function (_mm_extract_epi16): The compiler intrinsic equivalent of the function is

$$int\ \ \_mm\_extract\_epi16\ (\_\_m128i\ a,\ const\ int\ imm).$$

It extracts the 16 bit integer from the source operand (the first operand) determined by imm (the second operand).

Aggregation operations

SSE4.2 supports four aggregation operations which can be used to implement a wide range of string processing functions and can be applied on strings to process them simultaneously. The aggregation operations that can be used in string processing are equal each, equal any, ranges and equal ordered. We have applied the equal any operation to speed up processing of the text.

3. Problem Definition and Applications

Jumbled pattern matching problem [9, 14] (also known as Abelian pattern matching or permutation matching) is a variation of string matching. The problem is different from the classical string matching problem.

3.1 Problem Definition

Jumbled pattern matching addresses the problem of finding all permuted occurrences of a substring in a text. In other words, a substring u of T is a jumbled equivalent to P if the count of each character in P is equal to its count in u and $|P| = |u|$ holds. For instance, a permutation of a string $S = \texttt{abcbaab}$ occurs in $\texttt{cd}\underline{\texttt{aabbacb}}\texttt{dcabcdda}$ for $\Sigma = \{a, b, c, d\}$.

Jumbled pattern matching can be formalized with Parikh vectors [57]. The Parikh vector $p(S)$ of a string S over a finite alphabet is defined as the vector of multiplicities of the characters in S. For instance, a string $aaabddddbc$, can be represented in terms of a Parikh vector as (3,2,1,4) for $\Sigma = \{a, b, c, d\}$.

One of the variations of jumbled matching can be obtained by allowing edit distance between the text window and the pattern. In our algorithms, we address the k mismatch problem which is defined as follows. A string P' of length m is a k-approximate permutation of P, $0 \leq k < m$, if $|P'| = |P| = m$ holds and

$$\sum_{c \in set(P')} \max(cc(P', c) - cc(P, c), 0) \leq k,$$

where $set(P')$ is the set of characters in P' and $cc(u, c)$ is the number of occurrences of a character c in a string u. According to our definition, a 0-approximate permutation is an exact permutation. In other words, k-approximate permutation of P can also be defined as:

$$\sum_{c \in set(P')} |cc(P', c) - cc(P, c)| \leq 2k$$

Consider two strings P and P' each of length $m = 6$, having $k = 2$, such that, $P = aabbcc$ and $P' = abbbcc$. The above condition holds true, as the total difference between the character counts of each respective character (a, b, c) in P and P' is not greater than 4.

Our definition of approximate jumbled matching is different from the one presented by Burcsi et al. [9]. According to Burcsi et al., approximate jumbled matching is defined as follows. Consider s be a string of length n such that $s \in \Sigma^*$ and let $u = (u_0, \cdots, u_s)$ and $v = (v_0, \cdots, v_s)$ be two Parikh vectors, such that $u \leq v$ and $s = |\Sigma| - 1$ hold. Here $u \leq v$ means that $u_i \leq v_i$ for all $i \in \Sigma$ i.e. u is sub-parikh of v. The task is to find all maximal occurrences in s of some Parikh vector q, such that $u \leq q \leq v$. An occurrence of q is maximal (w.r.t. u and v) if neither $s_{i-1}, \cdots, s_j$ nor $s_{i-1}, \cdots, s_{j+1}$ is an occurrence of some Parikh vector q' such that $u \leq q' \leq v$.

Ejaz [20] considered three cost models for jumbled matching, substitution error model, insertion/deletion error model and minimum operations error model. The substitution model of Ejaz is equivalent to our k-approximate model.

For the given two strings S and $S^{'}$, insertion/deletion error model is defined as the minimum number of insertions of new characters in $S^{'}$ or deletion of existing characters of $S^{'}$ to transform $S^{'}$ into S. For instance, consider $S = aabbcc$ and $S^{'} = aabbbb$, then the InDel distance between S and $S^{'}$ is two, as two c's must be inserted in $S^{'}$ in place of two b's, to transform $S^{'}$ into S. Under this error model, the task is to find the approximate jumbled matches of a given pattern P of length m in a text T of length n.

The minimum operations error model is a combination of both the above mentioned models. In this model, both insertion/deletion and substitution of characters is allowed in the string, to transform it into another string.

3.2 Applications

Jumbled pattern matching has numerous applications in the field of bioinformatics, such as alignments of sequences [3], SNP discovery [5], discovery of repeated patterns [21], interpretation of mass spectrometry data [4]

and permutation pattern discovery in biosequences [21].

3.2.1 Gene Clustring

Sequencing of genome has become a regular practice in the last few decades, which in turn has led to the analysis of genomes at gene level [18, 62] and the correlation of genes. Genes having similar functionality are correlated to each other. Gene clustering [13, 33, 42, 55] helps to find the genes that are closely related to each other irrespective of the order in which they occur. Jumbled matching can be used to find the genes that are closely related to one another. Consistent occurrences of genes in the close proximity across genomes are believed to be functionally related. However, the order of the genes in chromosomes may be different. These kinds of gene clusters help solve the problem of local alignment of genes.

In the case of discovery of repeated patterns [21], jumbled matching algorithms can be used to solve the problem of local alignment of genes. However, the order of the occurrences of genes may not be the same. These usually can be modeled by Parikh fingerprints or character sets [6]. In other words, a group of genes that can appear in different orders in genomes, may be similar to one another. Genomes can also be modeled as strings or permutations of integers so that genes belonging to the same gene family are encoded by the same integer.

3.2.2 Composition Alignment

In composition matching [3], the main idea is to match the substrings that have similar composition or order. Composition alignment of two strings P and T having a scoring function $SF(p, t)$ is defined as the composition match between the substrings p and t of P and T, as well as the single character match between P and T. The task is to find the best scoring alignment.

For example, let $P = ACGTAGTACGT$ and $T = GCATGGGTCAG$ be two strings over the alphabet $\Sigma = (A, C, G, T)$. One possible composition alignment for P and T is:

$$\underline{G\,A\,C}\ \mathbf{T}\ \ \mathbf{G}\ \ \mathbf{T}\,T\,A\,T\,T\,\underline{C\,G\,T\,A}$$
$$\underline{C\,G\,A}\ \mathbf{T}\ \ \mathbf{G}\ \ \mathbf{T}\,G\,G\,G\,G\,\underline{A\,T\,G\,C}$$

Here, the characters in bold are used to depict exact single character

matches and substring composition matches are represented by underlined substrings. Note that composition matches can occur consecutively in an alignment.

Composition alignment is one of the major applications of jumbled pattern matching. In standard alignment of two strings, each character of the first string is matched to a single character of the second string. However, in composition alignment matching, the substrings that have the same characters are matched to the substring of the other string, even though the order of characters in the string can be different. It is easy to identify the subsequences that contain substrings of similar compositions. In other words, composition alignment is referred to as pairing of substrings of exactly matching composition, separated by insertions, deletions, or mismatches.

3.2.3 Mass spectrometry and SNP discovery

Jumbled pattern matching can also be used in the field of interpretation of mass spectrometer [4] to find strings with the same spectra. Mass spectra are simulated for every potential sequence and the resulting simulated spectra are then compared against the measured mass spectrum.

A single nucleotide polymorphism (SNP) is a variation at a single position in a DNA sequence among individuals. Each SNP represents a difference of a nucleotide in a single DNA. SNPs occur normally throughout a person's DNA. SNPs are believed to contribute strongly to the genetic variability in living beings. A comparatively new method to discover such polymorphisms is based on base-specific cleavage, where resulting cleavage products are analyzed by mass spectrometry. Simulating the mass spectrum that results from a base-specific cleavage experiment is relatively simple [5] and can be compared with simulating the mass spectrum of a protein.

3.2.4 Episode Matching

Mannila et al. [47] introduced the problem of episode matching. An episode is a collection of events that occur within a short time interval. This sequence of events needs to be analyzed in many areas of computer science, such as data mining, bioinformatics and machine learning. Episode matching helps to analyze the frequent occurring events and get significant information from the episodes. Jumbled matching is closely re-

lated to episode matching [60], such that the sub tasks of episode matching are similar to jumbled matching.

3.2.5 Other Applications

Other applications of jumbled pattern matching include string matching with a dyslectic word processor [10], table rearrangements [10], anagram checking [10], scrabble playing [10], common interval problem [34, 59, 61] etc.

4. Earlier Methods for JPM

In this chapter, we present previous algorithms for exact and approximate jumbled matching. Most algorithms process text by considering the window of m characters. If not otherwise stated, the approximate algorithms follow the substitution error model. We have divided the algorithms into four sections. In Section 4.1, we present algorithms that process the text window in the forward direction (left to right). Section 4.2 covers the algorithms that process the text window in the reverse direction (right to left). Section 4.1 and 4.2 deal with exact algorithms and approximate algorithms under the substitution model. Section 4.3 considers the approximate algorithms of JPM under advanced error models. Section 4.4 reviews shortly indexing methods for JPM.

4.1 Forward Methods

4.1.1 DC

Grossi & Luccio [30] presented the first algorithm for the approximate JPM problem. The original paper presents the algorithm as a filtering algorithm for approximate string matching, but the algorithm also solves the approximate JPM problem under the substitution model. Let us call Grossi & Luccio's approach DC (Distribution of Characters) [41]. The algorithm applies a queue of characters, which grows with the acceptable characters. The algorithm compares the distribution of characters in the pattern with the distribution of text characters in the queue. It is a forward type algorithm that scans the text from left to right and reports an occurrence of the permuted pattern in the text whenever the queue is full.

The pseudocode of the algorithm is presented in Alg. 1. An array C maintains the character counts of character c. In the preprocessing phase, an

element C_c is initialized to the number of occurrences of character c in the pattern. The value of C_c is the number of characters c that are missing from the queue. In the search phase, a variable Z is used as a counter for the mismatches among the substrings of the text against the prefixes of the pattern.

Algorithm 1 DC($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
   /* Preprocessing */
 1: for all c ∈ Σ do C_c ← 0
 2: for i ← 0 to m − 1 do
 3:       C_{p_i} ← C_{p_i} + 1
   /* Searching */
 4: front ← 0; rear ← −1; Z ← 0
 5: for j ← 0 to n − 1 do
 6:       c ← t_j
 7:       rear ← rear + 1
 8:       Q_rear ← c                                   /* Enqueue */
 9:       C_c ← C_c − 1
10:       if C_c < 0 then
11:             Z ← Z + 1
12:       while Z > k do
13:             c ← Q_front                            /* Dequeue */
14:             front ← front + 1
15:             if C_c < 0 then
16:                   Z ← Z − 1
17:             C_c ← C_c + 1
18:       if rear − front + 1 = m then
19:             occ ← occ + 1
20:             c ← Q_front                            /* Dequeue */
21:             front ← front + 1
22:             if C_c < 0 then
23:                   Z ← Z − 1
24:             C_c ← C_c + 1
25: return occ
```

For each character c of the text, an enqueue operation is performed and the value of the element C_c is decremented. If the value of C_c is negative, then counter Z is incremented. In the $while$ loop in line 12 of the algorithm, if the value of variable Z is greater that the error value k, the dequeue operation is performed until the value of Z is less than k. If the number of elements in the queue are equal to the size of the pattern, it means that an occurrence of the permuted pattern has been found. The window corresponding to the queue is moved forward by deleting the character that is located at the beginning of the queue. The next character of the text is inserted in the queue. The whole text is processed in a similar manner, reporting the total number of occurrences of the pattern in the

text.

Let us consider an example for the error value $k = 0$, a pattern P = $abab$ and a text T = $baabaabcab$ over an alphabet $\Sigma = \{a, b, c\}$. The execution begins with counting the number of occurrences of each character in the pattern and updating the array C ($C_a = 2$, $C_b = 2$ and $C_c = 0$) in the preprocessing phase. The character b at position t_0 of the text is assigned to the variable X and is inserted at the beginning of the queue.

The counter C_b is decremented to 1 and the next character a of the text is read at t_1. It is inserted in the queue and the counter C_a is decremented to 1. In a similar way, all characters until position t_4 are read updating the value of $C_a = -1$ and $C_b = 0$. As the condition in line 10 of the algorithm is true, the variable Z is incremented to 1. In line 12, the variable Z is compared against the error value k. If it is greater than k, then the character at the beginning of the queue is removed and the variable Z is decremented. As the condition for comparing the number of elements in the queue with the pattern length is true, an occurrence of the pattern in the text is reported. Continuing in the same manner, the whole text is scanned and all permuted occurrences of the pattern in the text are reported.

Analysis
The time complexity of DC depends on the *for* loop of the algorithm in line 5, taking $\Theta(n)$ time.

4.1.2 Count

The Count algorithm presented by Navarro [52] is another filtration method for approximate string matching. As the DC algorithm, it is also applicable to approximate JPM. The algorithm uses a sliding window approach in which a window of length m is slid from left to right throughout the text, maintaining the counts of characters in the current window and comparing them against the counts of characters in the pattern. Grabowsky et al. [29] presented an algorithm (called GFG) similar to Count for JPM.

Alg. 2 presents the pseudocode of the Count algorithm. In the preprocessing phase, a frequency counter C_c is initialized for character c to hold the number of occurrences of that character in the pattern. The variable $count$ holds the additive inverse of the number of wrong and extra charac-

ters in the alignment window of m characters. The initial value of $count$ is $-(m-k)$, which means that the window holds an occurrence of the pattern when $count \geq 0$. In the search phase, the array C maintains the character counts of the alignment window such that C_c is $cc(t_{i-m+1} \cdots t_i, c) - cc(P, c)$.

Algorithm 2 Count($P = p_0 p_1 \cdots p_{m-1}, T = t_0 t_1 \cdots t_{n-1}, k$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do C_c ← 0
 2: for i ← 0 to m − 1 do
 3:     C_{p_i} ← C_{p_i} + 1
    /* Searching */
 4: count ← −(m − k)
 5: for i ← 0 to m − 1 do
 6:     if C_{t_i} > 0 then
 7:         count ← count + 1
 8:     C_{t_i} ← C_{t_i} − 1
 9: while i < n do
10:     if count ≥ 0 then
11:         occ ← occ + 1
12:     C_{t_{i−m}} ← C_{t_{i−m}} + 1
13:     if C_{t_{i−m}} > 0 then
14:         count ← count − 1
15:     if C_{t_i} > 0 then
16:         count ← count + 1
17:     C_{t_i} ← C_{t_i} − 1
18:     i ← i + 1
19: return occ
```

Before processing the whole text, the first window is processed. The character counts for each character c of the first alignment window are decremented from C_c (in step 8), also, the variable $count$ is updated. Each time the window is moved forward, a new character of the text t_i is included and the leftmost character of the previous window at position t_{i-m} is excluded. For each character c that is excluded from the text window, value of C_c is incremented by 1. If the value of C_{t_i} is greater than zero, it signifies that the character is present in the pattern, so the variable $count$ is incremented. If the value of element $C_{t_{i-m}}$ contains a positive value, it signifies that the character was in the pattern, so variable $count$ is decremented. If at any given position i, $m-k$ or more characters of the pattern are in the text window, the window contains an occurrence of the permuted pattern in the text.

Let us consider an example for a pattern P = $abba$ and a text T = $cbabacbabc$ over an alphabet $\Sigma = \{a, b, c,\}$ with $k = 0$. In the preprocessing phase, the array C is initialized with the number of occurrences of each

character in the pattern i.e. $C_a = 2$, $C_b = 2$ and $C_c = 0$. The variable $count$ is initialized with the value $-(m-k)$ i.e -4. The array C is updated with the character counts of the first alignment window of the text $[cbab]acbabc$. The variable $count$ receives the value -1.

The execution continues from the leftmost character of the next text window ($c[baba]cbabc$). In that window, the character counts are the same as in the pattern, and the algorithm reports a match. The execution continues by moving the the text window to the right by one position. In the same way, the whole text is processed and the total number of occurrences of the permuted pattern is reported at the end.

Analysis
The complexity of the Count algorithm is determined by the main $while$ loop in line 9. The algorithm $count$ works linearly since it processes the whole text by shifting the window by one position yielding a complexity of $\Theta(n)$.

4.1.3 Mcount

Besides a single pattern algorithm, Navarro [52] also presented a multi-pattern variation for patterns of equal length called Mcount. Each character has a count variable of its own, and a field of $d+1$ bits is allocated for each pattern. The most significant bit of each field is a kind of overflow bit. The bitvector E_c holds a field of $d+1$ bits for each pattern. The initial values of fields in E_c and C are $2^d + cc(P_j, c) - 1$ and $2^d - (m-k)$, respectively, for the jth pattern.

The bit mask I holds one in the least significant bit of every field in E_c. The operation $(E_c >> d)\,\&\,I$ extracts the most significant bit of E_c. The condition $C \geq 0$ means that at least one overflow bit is set in C, i.e. $m-k$ acceptable characters of at least one pattern have been found. In the case of a single pattern, this is enough to recognize an occurrence. In the case of two or more patterns, a verification step is required because the condition $C \geq 0$ does not specify which pattern matches the text window.

The pseudocode of Mcount is presented in Alg. 3 adapted for the case of single pattern. In the preprocessing phase, the offset d is given the value $\lceil log\ m \rceil$.

Algorithm 3 Mcount($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Preprocessing */
 1: d ← ⌈log_2 m⌉
 2: for all c ∈ Σ do E_c ←~ 0 >> w − d
 3: for j ← 0 to m − 1 do
 4:       E_{p_j} ← E_{p_j} + 1
 5: C ← (1 << d) − (m − k)
    /* Searching */
 6: j ← 0; occ ← 0; I ← 1
 7: while j < m do
 8:       c ← t_{j+1}
 9:       C ← C + ((E_c >> d) & I)
10:       E_c ← E_c − 1
11:       j ← j + 1
12: while j < n do
13:       if C ≥ 0 then
14:             occ ← occ + 1
15:       E_{t_{j−m}} ← E_{t_{j−m}} + I
16:       C ← C − (E_{t_{j−m}} >> d) & I
17:       C ← C + (E_{t_j} >> d) & I
18:       E_{t_j} ← E_{t_j} − I
19:       j ← j + 1
20: return occ
```

4.1.3.1 Analysis

The main $while$ loop of the algorithm is in line 12, yielding a complexity of $\Theta(n)$. For a single pattern, Mcount is faster than Count (probably because if statements are relatively slower in modern processors), as it contains only one if statement in the main loop, whereas Count contains three if statements in the main loop.

4.1.4 PBA

Ejaz presented the PBA (Prefix Based Algorithm) [20] [1] algorithm for exact JPM that works by processing the characters of the text window in the prefix manner. The current window of the text is searched for the permutation of the pattern. The frequencies of each character in the text window and the pattern are compared, and the text window is shifted by one character to the right. A variable M is used to keep track of the count of mismatch characters in the text window.

The central idea of this algorithm is to shift the text window by one position to the right and compare the rightmost character of the current text window and the leftmost character of the previous window, and up-

[1] Ejaz did not mention DC and Count algorithms in his dissertation.

date the variable M if the count of two characters is not the same. The whole text is scanned in the same manner from left to right, moving one position at one step. This algorithm is a variation of the Count algorithm. The major difference is that it maintains two arrays to compare the frequencies of the characters in the text window and the pattern by adding a new character of the current text window and removing one character from the previous window. In contrast, the Count algorithm examines the frequency of characters in a text window using the same array.

Algorithm 4 PBA($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do A_c ← 0
 2: for i ← 0 to m − 1 do
 3:       A_{p_i} ← A_{p_i} + 1
    /* Searching */
 4: for all c ∈ Σ do C_c ← 0
 5: for j ← 0 to m − 1 do
 6:       C_{t_i} ← C_{t_i} + 1
 7: M ← 0
 8: for all i ∈ Σ do
 9:       if C_i ≠ A_i then
10:             M ← M + 1
11: if M = 0 then
12:       occ ← occ + 1
13: i ← 0
14: while i ≤ n − m do
15:       if t_i ≠ t_{i+m} then
16:             C_{t_i} ← C_{t_i} − 1
17:             if C_{t_i} = A_{t_i} then
18:                   M ← M − 1
19:             else
20:                   if C_{t_i} = A_{t_i} − 1 then
21:                         M ← M + 1
22:             C_{t_{i+m}} ← C_{t_{i+m}} + 1
23:             if C_{t_{i+m}} = A_{t_{i+m}} then
24:                   M ← M − 1
25:             else
26:                   if C_{t_{i+m}} = A_{t_{i+m}} + 1 then
27:                         M ← M + 1
28:       if M = 0 then
29:             occ ← occ + 1
30:       i ← i + 1
31: return occ
```

Alg. 4 represents the pseudocode of the algorithm. Two arrays C and A are used to count the frequencies of each character in the text window and the pattern respectively. The value of the variable M shows the difference in the counts of characters arrays A and C. In the preprocessing phase, the

array A is initialized with the frequencies of all characters in the pattern. The search phase begins by updating the array C with the frequencies of the first m characters of the text. The variable M is initialized to zero in the beginning. For each character of the text window, the array A is compared against the frequency of each character in the array C, and the variable M is updated.

The value of the variable $M = 0$, indicates that the frequency of each character c in the current text window (t_i to t_{i+m-1}), is the same as the frequency of that character in the pattern (i.e., each C_c equals A_c). The window is moved by one position to the right and the vector C is updated. For instance, if the leftmost character x of the previous window ($T = \ldots x t_i + 1 \ldots t_{i+m-1} y \ldots$) is not present in the current window, this means it is different from the new character y that is added to the current window. The frequency for x is decremented if the current window does not contain the leftmost character of the previous window.

The content of the element C_x is not updated if the character x is the same as y. Otherwise, the differences between the frequencies of x in both the arrays (A and C) are inspected after moving the text window towards the right by one position. A difference of zero indicates that the frequency of x in the array C equals the frequency of x in the array A and variable M is decremented. A difference of one indicates that the frequency of respective elements in C and A differ by one, hence the value of M is incremented. The whole algorithm works in the same manner, and the total number of occurrences of the permuted pattern in the text is reported at the end.

For example, let the pattern P be $aabc$ and the text T be $cbabacbabc$ over an alphabet $\Sigma = \{a, b, c, \}$. The array A is updated with the value $A_a = 2$, $A_b = 1$ and $A_c = 1$ and array C is updated with the character counts of the first window ($[cbab]acbabc$) $Ca = 1$, $C_b = 2$ and $Cc = 1$. The processing of the text begins by comparing the counts of characters in the first text window and the pattern. As the values of C and A for characters a and b are different, no occurrence of the permuted pattern is reported. In the next step, the leftmost character ($[\underline{c}bab]\underline{a}cbabc$) of the current window is compared against the character next to the window and the variable C_c gets value 0 and M has value 1. The values of C and A are compared for each character of the window and the value of M is updated. The window is moved forward by one position. The whole text is scanned in this man-

ner, reporting total number of occurrences of the permuted pattern in the text.

Analysis
The main loop in this algorithm is the $while$ loop in line 14. PBA works linearly, yielding the complexity of $\Theta(n)$.

4.1.5 ASM

The ASM (Approximate abelian pattern matching under the Substitution error Model) algorithm [20] finds the approximate permutation matches of a pattern in the text, using the substitution error model. The processing starts by setting a window of length m at the beginning of the text. Based on the substitution error model, a distance is calculated between the pattern and the current window. If the distance is less than the error k, then there is an occurrence of a permuted pattern in the text window. The window is slid from left to right by one position to find another occurrence of the permuted pattern. This algorithm is a variation of the Count algorithm, by calculating the number of substitutions required to transform the current text window into the permuted pattern.

The pseudocode of the algorithm is presented in Alg. 5. In the preprocessing phase, an array C is initialized for the m characters of the pattern. In the search phase, the substitution distance between the current window and the pattern is calculated. The computation required to find whether the current text window contains a permutation of the pattern, is half of the sum of the absolute differences in the frequencies of the present text window and the pattern. If the value of the above computation is less than or equal to k, then there is an occurrence of the permuted pattern in the text window. At each step, the characters at position t_i and t_{i+m} are compared. A difference leads to the update of the vector C_{t_i} and $C_{t_{i+m}}$ ($C_{t_i} = C_{t_i} - 1$ and $C_{t_{i+m}} = C_{t_{i+m}} + 1$).

The value of the vector C, determines the number of substitutions. If the number is less than or equal to k, an occurrence of the permuted pattern is reported. The text window is moved forward by one position and the vector C is updated if the newly added character at the position t_{i+m} is different from the t_i. Hence, the value of $subs$ is updated depending on the value of the vector C. The whole text is processed in the same way and all starting positions of the permuted patterns in the text are reported in the

end.

Let us consider an example with a pattern $P = aabbc$ and a text $T = caaabacabcabc$ over an alphabet $\Sigma = \{a, b, c,\}$ for the error value $k = 1$. The values of the elements are $C_a = -2, C_b = -2$ and $C_c = -1$. As the variable $subs$ stores the number of errors in the current window, it has value 2. Thereafter, $subs$ is updated with the value 1 (in line 8) for the first window $[caaab]acabcabc$. Since $subs$ is equal to k, there is a match in the first window of the text. The window is slid by one position through the whole text, updating the value of the vector C and $subs$, and reporting the total number of occurrences of the permuted pattern in the text.

Algorithm 5 ASM($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do C_c ← 0
 2: for i ← 0 to m − 1 do
 3:       C_{p_i} ← C_{p_i} − 1
    /* Searching */
 4: for j ← 0 to m − 1 do
 5:       C_{t_j} ← C_{t_j} + 1
 6: subs ← 0
 7: for all c ∈ Σ do subs ← subs + |C_c|
 8: subs ← subs/2
 9: if subs ≤ k then
10:       occ ← occ + 1
11: i ← 0
12: while i ≤ n − m do
13:       if t_i ≠ t_{i+m} then
14:             C_{t_i} ← C_{t_i} − 1
15:             C_{t_{i+m}} ← C_{t_{i+m}} + 1
16:             if C_{t_i} < 0 then
17:                   if C_{t_{i+m}} > 0 then
18:                         subs ← subs + 1
19:       else
20:             if C_{t_{i+m}} ≤ 0 then
21:                   subs ← subs − 1
22:       if subs ≤ k then
23:             occ ← occ + 1
24:       i ← i + 1
25: return occ
```

Analysis

In this algorithm, the main $while$ loop is in line 12, yielding a complexity $\Theta(n)$.

4.2 Backward Methods

In this section, we explain some of the previous solutions that use the backward scanning approach for processing the text window.

4.2.1 BAM

BAM (Bit-parallel Abelian Matcher) [11] is an exact JPM algorithm presented by Cantone and Faro that applies the concept of bitparallelism and backward scanning of the alignment window of the text. The characters inside the text window are searched for a permutation of the pattern. The algorithm works by sliding a window of size m over the whole text from left to right. The text window is shifted to the right of the text character where a mismatch is reported.

Algorithm 6 BAM($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do C_c ← 0; M_c ← 0
 2: for i ← 0 to m − 1 do
 3:         C_{p_i} ← C_{p_i} + 1
 4: logm ← ⌊log_2 m⌋
 5: sh ← 0; I ← 0; F ← 0
 6: for all i ∈ Σ do
 7:         if C_i ≥ 0 then
 8:                 M_i ← M_i | (1 << sh)
 9:                 I ← I | (((1 << logm) − C_i − 1) << sh)
10:                 F ← F | (1 << (sh + logm))
11:                 sh ← sh + logm + 2
12: F ← F | (1 << sh)
13: for all i ∈ Σ do
14:         if C_i = 0 then
15:                 M_i ← M_i | (1 << sh)
    /* Searching */
16: s ← 0
17: while s ≤ n − m do
18:         D ← I; j ← s + m − 1
19:         while j ≥ s do
20:                 D ← D + M_{t_j}
21:                 if D & F then
22:                         break
23:                 j ← j − 1
24:                 if j < s then
25:                         occ ← occ + 1
26:                         s ← s + 1
27:                 else
28:                         s ← j + 1
29: return occ
```

The central idea of this algorithm is to associate a counter with each distinct character of the pattern. In addition to that, a counter of one bit is reserved for the remaining characters of the alphabet which do not occur in the pattern.

A field of $g(c) = \lfloor \log cc(P, c) \rfloor + 2$ bits is reserved for each character c appearing in P. As in Mcount, the most significant bit of each field is a kind of overflow bit. The initial value of the field is $2^{g(c)-1} - cc(P, c) - 1$ which means that $cc(P, c) + 1$ occurrences trigger the overflow bit. The adaptive width of the bit fields makes it possible to handle longer patterns than with a fixed width.

The pseudocode of BAM is presented in Alg. 6. In the original article, the corresponding pseudocode is only for a fixed length field for the characters. The algorithm allocates $\lfloor \log m \rfloor + 2$ bits for each character in the pattern. In the preprocessing phase, a bit mask M_c is computed for each distinct character that appears in the pattern and D is the state vector. M_c signifies the bits associated with character c that can be added from the current text window to D for each occurrence of c in the text window. This bit mask is used to update D. The bitvector I holds the initial values of all fields corresponding to the characters that appear in the pattern, and the bitvector F sets each overflow bit.

In the search phase, the whole text is scanned and at the beginning of processing of each text window, the state vector D is initialized by the vector I. The characters are scanned from right towards left within the text window. For each processed character c of the text, D is incremented to $D + M_c$. The state vector D is tested with the overflow vector F to detect if any of the overflow bits is set or not. This update step halts when an overflow bit in D is set or when the left end of the window is reached. The window is moved forward after the location of the overflow character for the next alignment.

Let us consider an example having a pattern $P = abbccc$ and a text $T = acbbaacbcbccbac$ over an alphabet $\Sigma = \{a, b, c\}$. The values of array M_a, M_b, M_c are represented in the tables below. The values of the vectors D, I and F are as follows.

The leftmost bit of the vector D and other vectors represents an overflow bit for the character that is not present in the text window. Below, the counters associated with each character are separated by a vertical line. The leftmost field marked by x is an overflow bit for characters not

present in the pattern. It is to be noted that the corresponding field is marked by x also in the subsequent examples. The rightmost counter is associated with the character of lowest value (character a in this example) and the leftmost counter is associated with the character of highest value (character c in this example).

	x	c				b				a			
M_a	**0**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	1
M_b	**0**	**0**	0	0	0	**0**	0	0	1	**0**	0	0	0
M_c	**0**	**0**	0	0	1	**0**	0	0	0	**0**	0	0	0

	x	c				b				a			
I	**0**	**0**	1	0	0	**0**	1	0	1	**0**	1	1	0
F	**1**	**1**	0	0	0	**1**	0	0	0	**1**	0	0	0
D	**0**	**0**	1	0	0	**0**	1	0	1	**0**	1	1	0

While reading the first text window $([acbbaa]cbcbccbac)$, the characters are scanned from right to left. The first (rightmost) character of the current text window to be read is a. The array D is updated with the operation $D+M_a$ and the updated value is represented as follows. After reading the next character a at position t_4, an overflow occurs in the array D. The test $D \ \& \ F$ succeeds as the overflow bit corresponding to the character a is set in the state vector D and it breaks the loop. The execution skips the text until the location of the overflow character is encountered. The window is shifted to a new position at t_5 $(acbba[acbcbc]cbac)$.

	x	c				b				a			
$D + M_a$	**0**	**0**	1	0	0	**0**	1	0	1	**0**	1	1	1

	x	c				b				a			
$D + M_a$	**0**	**0**	1	0	0	**0**	1	0	1	**1**	1	1	1

	x	c				b				a			
D	**0**	**0**	1	0	0	**0**	1	0	1	**1**	1	1	1
F	**1**	**1**	0	0	0	**1**	0	0	0	**1**	0	0	0
$D \ \& \ F$	**0**	**0**	0	0	0	**0**	0	0	0	**1**	0	0	0

The state vector D is initialized by the vector I as represented below. The next character to be read is c at position t_{10}. The array D is updated by the vector M while each character in the text window is read and tested with the vector F for any overflow. As there is no overflow in this window, the execution reaches the leftmost position of the window.

	x		c			b				a			
D	**0**	**0**	1	0	**0**	**0**	1	0	1	**0**	1	1	0

The state vector D is tested against the vector F as shown below. No overflow is reported and an occurrence of the jumbled pattern in the current text window is reported. In the next step, the window is moved forward by one step and once again, the value of the vector D is initialized by the vector I. In this manner, the whole text is processed and all occurrences of the jumbled pattern in the text are reported.

	x		c			b				a			
D	**0**	**0**	1	1	1	**0**	1	1	1	**0**	1	1	1
F	**1**	**1**	0	0	0	**1**	0	0	0	**1**	0	0	0
D & F	**0**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	0

Analysis
We assume that the bit vector D fits to the computer word. The complexity of BAM depends on the main $while$ loop of the algorithm in line 17 and the inner $while$ loop in line 19. This algorithm works in $O(nm)$ time in the worst case, as it executes both outer and inner loops. The algorithm takes $O(\frac{n}{m})$ time in the best case.

4.2.2 SBA

The SBA (Suffix Based Algorithm) [20] algorithm introduced by Ejaz, works by reading the characters from right to left in the search window of the text. The algorithm compares the frequency of the characters of the text window and the pattern. A mismatch among the frequencies indicates the absence of permuted pattern in the text window and the window is moved forward after the location of a mismatch character.

The pseudocode of the algorithm is presented in Alg. 7. The arrays C and A store the number of occurrences of each character of the text window and the pattern respectively. While reading the characters in the current window, if an overflow of frequency in C occurs, further scanning is not required as the current window does not contain any permutation of the pattern. A list is maintained to keep the distinct characters that are read in a window. When an overflow occurs, the value of those elements of C which are in the list are set to zero.

When there is no overflow while reading the characters of the window, the window is referred to as a safe window. As soon as there is an occur-

rence of a safe window, the starting position of the window is reported as a location for the match. If the window is not a safe window, the pointer gets updated to the starting position of the window next to the overflow character.

Algorithm 7 SBA($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}$)

```
   /* Preprocessing */
 1: for all c ∈ Σ do A_c ← 0
 2: for i ← 0 to m − 1 do
 3:       A_{p_i} ← A_{p_i} + 1
   /* Searching */
 4: for all c ∈ Σ do C_c ← 0
 5: i ← 0
 6: while i ≤ n − m do
 7:       overflow ← 0; j ← i + m − 1
 8:       while j ≥ i and overflow = 0 do
 9:             C_{t_j} ← C_{t_j} + 1
10:             if C_{t_j} = 1 then
11:                   insert t_j in list
12:             if C_{t_j} > A_{t_j} then
13:                   overflow ← 1
14:             j ← j − 1
15:       if overflow = 0 then
16:             occ ← occ + 1
17:       i ← j + 2
18:       for all c ∈ list do C_c ← 0; remove c from list
19: return occ
```

Let us consider an example with a pattern $P = acca$ and a text $T = caaaccbabacb$ over an alphabet $\Sigma = \{a, b, c\}$. The execution starts by scanning the first text window $caaa$ from right towards left. As the algorithm reads the character a at t_1 ($c\underline{a}aa$), the frequency of a in the current window is updated to 3. As the frequency does not match with the count of a in the pattern, the window is moved after the mismatch character. The new window is $ca[aacc]babacb$. In this window, there is a match and it is reported at line 15 of the algorithm, and the window is moved forward by one position. Continuing in the same way, the whole text is scanned by moving forward the text window and reporting the total number of matches.

Analysis

The complexity of Alg. 7 depends on the main $while$ loop of the algorithm in line 6. This $while$ loop has an inner $while$ loop in line 8. The best case complexity of the algorithm is $O(\frac{n}{m})$ and has $O(nm)$ worst-case time complexity.

4.3 Approximate Methods for Advanced Error Models

In this section, we explain the previous solutions for approximate JPM under advanced error models.

4.3.1 AIDM

The AIDM (Approximate abelian pattern matching under the Insertion/Deletion (InDel) error Model) algorithm presented by Ejaz [20], finds the approximate matches for the jumbled pattern matching by performing the insertions and/or deletions in the current text window in order to make it equal to the permuted pattern P, depending on the error limit k.

Algorithm 8 AIDM($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Preprocessing */
 1: for all i ∈ Σ do C_i ← 0
 2: for i ← 0 to m − 1 do
 3:       C_{P_i} ← C_{P_i} − 1
    /* Searching */
 4: for i ← 0 to m − k − 1 do
 5:       C_{t_i} ← C_{t_i} + 1
 6: ins ← 0; del ← 0
 7: for all i ∈ Σ do
 8:       if C_i > 0 then
 9:             del ← del + C_i
10:       else
11:             if C_i < 0 then
12:                   ins ← ins + |C_i|          /* |C_i| is absolute value of C_i */
13: E ← 0
14: if del ≤ k then
15:       match ← Check(t, C, ins, del, k, m − k − 1)
16:       if match > E then
17:             occ ← occ + 1; E ← match
18: i ← 1
19: while i ≤ n − m + k do
20:       if t_{i−1} ≠ t_{i+m−k−1} then
21:             C_{t_{i−1}} ← C_{t_{i−1}} − 1
22:             C_{t_{i+m−k−1}} ← C_{t_{i+m−k−1}} + 1
23:             if C_{t_{i−1}} ≤ 0 then
24:                   if C_{t_{i+m−k−1}} > 0 then
25:                         ins ← ins + 1; del ← del + 1
26:       if C_{t_{i+m−k−1}} ≤ 0 then
27:             ins ← ins + 1; del ← del + 1
28:       if del ≤ k then
29:             match ← Check(t, C, ins, del, k, m − k − 1)
30:             if match > E then
31:                   occ ← occ + 1; E ← match
32:       i ← i + 1
33: return occ
```

In this algorithm, a window of length $m - k$ is slid from left to right along the text where k is the error value. If the substring is a potential match, then the processing is extended towards the right. The processing continues until the extended window is not a potential match. A potential match is a substring $S = t_i \dots t_j$ if there exists $j' > j$ such that a substring $S' = t_i \dots t_{j'}$ with $j' > j$ can be an approximate match. A potential match cannot be longer than $m + k - 1$. Keeping the longest approximate match, if this is also a maximal match (if the end position of the latest match is greater than the previous match), then the starting position of the current window is marked to be considered for an actual match. Moving forward this way, all positions for the potential matches are marked.

A function $Check$ is invoked from the algorithm in line 15 when the current text window contains a potential match. It returns the ending position of the longest match or it returns zero if no match is found. The pseudocode of the function is presented in algorithm 9. If the characters in the current window are different from the characters of the previous window, the variables ins and del are updated. The current window contains the potential match, and $Check$ is invoked to find whether the value returned by the function is greater than E. If the returned value by the function $Check$ is greater than the maximal match E, then a match is found and the value of E is changed to the current $match$ value. The whole scanning phase is repeated until all occurrences of the permuted pattern are reported in the text.

Analysis

The main loop of this algorithm is in line 19. The time complexity of the algorithm is $O(n+Mk)$ [20], where M is the number of potential matches, as the function $Check$ is invoked M times during the execution of the algorithm. The time complexity of $Check$ is $O(k)$.

Algorithm 9 Check(t, C, ins, del, k, u)

```
1: match ← 0
2: if ins + del ≤ k then
3:     match ← u
4: u' ← u; F ← True
5: while F do
6:     u' ← u' + 1
7:     C_{t_{u'}} ← C_{t_{u'}} + 1
8:     if C_{t_{i-1}} ≤ 0 then
9:         ins ← ins − 1
10:    else
11:        del ← del + 1
12:    if ins + del ≤ k then
13:        match ← u'
14:        if del > k then
15:            F ← False
16:    while u' > u do
17:        C_{t_{u'}} ← C_{t_{u'}} − 1
18:        u' ← u' − 1
19: return match
```

4.3.2 Matching with Query Bounds

P. Burcsi, F. Cicalese, G. Fici & Zs. Lipták [8, 9] presented solutions on the approximate jumbled pattern matching using query bound u, v to find out the maximal occurrences of a Parikh vector in a string s. The query bounds u and v are used to form an upper bound on the Parikh vector q, such that $u \leq q \leq v$. The problem has been solved by constructing a wavelet tree [31] of a string s in the preprocessing phase. For example, let s be a string, such that $s = abbcabab$ and $q = (1, 2, 1)$ satisfies query bounds u and v, having values $(1, 2, 1)$ and $(2, 2, 2)$ respectively. The Parikh vector q has three maximal occurrences in the string s ($abbc$, $bbca$ and $bcab$).

Another solution for this problem has been introduced by Castellanos et al. [12], using the concept of expand, shrink and refine to move two pointers L and R. The pointers L and R represent a window pointing the potential positions where an occurrence of the approximate permuted pattern can be found. It uses the sliding window approach to move two pointers, which depends on the sub-parikh and super-parikh vectors. Let p and q be two Parikh vectors, such that $p \leq q$ then, p is a sub-parikh of q and q is super-parikh of p.

In the expansion phase, a window that is placed before the first character of a string s is extended towards the right until its corresponding Parikh vector is a super-parikh vector of u. In the case of shrinking phase,

the window shrinks from the left side until it contains the sub-parikh vector of v. Castellanos & Pinzón [12] also presented a variation of approximate jumbled pattern matching known as δ approximate jumbled pattern matching, where a vector δ is considered as a vector of possible errors.

4.4 Indexing

The task of indexing for jumbled matching addresses the problem of indexing a string T over an alphabet Σ for a given pattern P, and to find all substrings of S that are permutations of P. In other words, a given text T is preprocessed so that a query $Q(T, P)$ can be solved, where $Q(T, P)$ returns true if and only if any permutation of P occurs in T as a substring. Indexing a string for jumbled matching is more difficult to solve than in other variations of string matching.

The indexing problem is more challenging than the jumbled matching itself. Moosa et al. [49] presented an index for jumbled pattern matching for binary alphabets. Kociumaka et al. [44] introduced a solution for any constant sized alphabet. Various solutions [1, 10, 26, 28] to this problem have been proposed in the recent past.

5. New Methods for Exact JPM

In this chapter, we introduce new algorithms for exact jumbled matching. Most of the algorithms apply bitparallelism. The main idea behind these algorithms is to associate counters with the characters in the pattern. These algorithms scan the whole text and update the counters associated with the characters present in the text window and report the occurrences of the permuted pattern in the text.

We present six new algorithms DBAM, BAMs, BAM2, EBL, EFS and EFB. The first four algorithms work by backward scanning of the text window, and the remaining algorithms process the text window in a forward manner.

5.1 DBAM

DBAM (Dynamic BAM) is a dynamic variation of BAM [11]. As stated before, the article of BAM [11] suggested dynamic fields, but the given algorithm allocates a field of the same width for each character in the pattern. Because the implementation is not trivial, we present it here. The pseudocode for the dynamic allocation of the bits is shown in Alg. 10. We use $\lfloor \log x \rfloor + 2$ bits for each character appearing x times in the pattern.

Let us consider an example having a pattern $P = gacgcg$ and a text $T = acgtggcagcagctc$ over an alphabet $\Sigma = \{a, c, g, t\}$, the values of M_a, M_c, M_g, M_x are represented below where M_a, M_c, M_g are the occurrence vectors for the characters a, c, g respectively and M_x is the occurrence vector for the characters that are not present in the pattern.

Algorithm 10 DBAM($P = p_0 p_1 \cdots p_{m-1}$)

/* Preprocessing */
1: **for all** $c \in \Sigma$ **do** $C_c \leftarrow 0; M_c \leftarrow 0$
2: **for** $j \leftarrow 0$ **to** $m-1$ **do**
3: $\quad C_{p_j} \leftarrow C_{p_j} + 1$
4: $sh \leftarrow 0; I \leftarrow 0; F \leftarrow 0$
5: **for all** $i \in \Sigma$ **do**
6: $\quad$ **if** $C_i > 0$ **then**
7: $\quad\quad M_i \leftarrow M_i \mid (1 << sh)$
8: $\quad\quad logp \leftarrow \lfloor log(C_i) \rfloor$
9: $\quad\quad I \leftarrow I \mid (((1 << (logp+1)) - C_i - 1) << sh)$
10: $\quad\quad F \leftarrow F \mid (1 << (sh + (logp+1))); sh \leftarrow sh + logp + 2$
11: $F \leftarrow F \mid (1 << sh)$
12: **for all** $j \in \Sigma$ **do**
13: $\quad$ **if** $C_j = 0$ **then**
14: $\quad\quad M_j \leftarrow M_j \;\&\; (1 << sh)$

	x		g			c			a	
M_a	**0**	**0**	0	0	**0**	0	0	**0**	1
M_c	**0**	**0**	0	0	**0**	0	1	**0**	0
M_g	**0**	**0**	0	1	**0**	0	0	**0**	0
M_x	**1**	**0**	0	0	**0**	0	0	**0**	0

The values of initial vector I, overflow vector F and state vector D are represented as follows. The leftmost bit of the vectors represents the overflow bit. The counters associated with each character are separated by a vertical line.

	x		g			c			a	
I	**0**	**0**	0	0	**0**	0	1	**0**	0
F	**1**	**1**	0	0	**1**	0	0	**1**	0
D	**0**	**0**	0	0	**0**	0	1	**0**	0

The execution begins by processing the first text window ($[acgtgg]cagcagctc$). The characters are processed from right to left inside the text window. The first character to be read is g ($[acgtg\underline{g}]cagcagctc$) at position t_5. Array D is updated with the operation $D + M_g$, as shown below. After reading the next character g at position t_4, the algorithm reads character t. As the character t is not present in the pattern, an overflow bit for the remaining characters is marked as set in the array D.

	x		g			c			a	
D	**0**	**0**	0	0	**0**	0	1	**0**	0
M_g	**0**	**0**	0	1	**0**	0	0	**0**	0
D	**0**	**0**	0	1	**0**	0	1	**0**	0

	x		g			c			a	
D	**0**	**0**	0	1	**0**	0	1	**0**	0
M_g	**0**	**0**	0	1	**0**	0	0	**0**	0
D	**0**	**0**	1	0	**0**	0	1	**0**	0

	x		g			c			a	
D	**0**	**0**	1	0	**0**	0	1	**0**	0
M_t	**1**	**0**	0	0	**0**	0	0	**0**	0
D	**1**	**0**	1	0	**0**	0	1	**0**	0

In the next step, the test of $D \ \& \ F$ succeeds (shown below) as the overflow bit is set and it breaks the loop.

	x		g			c			a	
$D \ \& \ F$	**1**	**0**	0	0	**0**	0	0	**0**	0

The execution skips the remaining characters to be scanned in the text window. The text window is shifted after the position t_3 in $acgt[ggcagc]agctc$. The execution continues at the position t_9. The character to be read is c in $acgt[ggcag\underline{c}]agctc$. The array D is updated and tested for any overflow for each character in the text window.

No overflow is reported in the current text window. The execution reaches the beginning of the current window without setting any overflow bit in the state vector D. The array D is tested against the array F, as shown below. There is no overflow and the occurrence of the permuted pattern in the current text window is reported.

	x		g			c			a	
D	**0**	**0**	1	1	**0**	1	1	**0**	1
F	**1**	**1**	0	0	**1**	0	0	**1**	0
$D \ \& \ F$	**0**	**0**	0	0	**0**	0	0	**0**	0

In the next step, the window is moved forward by one position at location t_5 $(acgtg[gcagca]gctc)$. The value of the vector D is initialized by the

vector I. In this manner, the whole text is scanned by shifting the window, and the total number of occurrences of the jumbled pattern in the text is reported.

5.2 BAMs

BAMs is a variation of the BAM algorithm, where one counter a.k.a bin is shared. We introduced the shared bin variation of BAM, considering the fact that, if the pattern is long, w (the number of bits in the computer word) bits are not enough to hold a distinct bin for each character appearing in the pattern. Distinct characters for bins are selected from the pattern in the left-to-right order. When w bits is exceeded, the character in turn and rest of the characters share the last fitting bin. This heuristic does not necessarily lead to an optimal solution. This is a kind of alphabet reduction. Then, instead of "report occurrence", each match candidate should be verified. Otherwise, BAMs works like BAM.

Algorithm 11 BAMs($P = p_0p_1 \cdots p_{m-1}$)

/* Preprocessing */
for all $c \in \Sigma$ **do** $C_c \leftarrow 0; M_c \leftarrow 0$
for $j \leftarrow 0$ **to** $m-1$ **do**
    $C_{p_j} \leftarrow C_{p_j} + 1$
$sh \leftarrow 0; l \leftarrow 0; F \leftarrow 0; Flag \leftarrow false$
for all $i \in \Sigma$ **do**
    if $C_i > 0$ **then**
        $M_i \leftarrow M_i \mid (1 << sh); logp \leftarrow \lfloor log(C_i) \rfloor$
        $logm \leftarrow \lfloor log(m - C_i) \rfloor$
        if $sh + logp + (logm + 5) > w$ **then**
            $Flag \leftarrow true; goto\ \ rest$
        $I \leftarrow I \mid (((1 << (logp + 1)) - C_i - 1) << sh)$
        $F \leftarrow F \mid (1 << (sh + (logp + 1))); sh \leftarrow sh + logp + 2$
rest:
if $Flag$ **then**
    $count \leftarrow C_i$
    for $j \leftarrow i + 1$ **to** $|\Sigma| - 1$ **do**
        if $C_j > 0$ **then**
            $M_j \leftarrow M_j \mid (1 << sh)$
            $count \leftarrow count + C_j$
    $logp \leftarrow \lfloor log(count) \rfloor$
    $I \leftarrow I \mid (((1 << (logp + 1)) - C_i - 1) << sh)$
    $F \leftarrow F \mid (1 << (sh + logp + 1)); sh \leftarrow sh + logp + 2$
$F \leftarrow F \mid (1 << sh)$
for all $i \in \Sigma$ **do**
    if $C_i = 0$ **then**
        $M_i \leftarrow M_i \,\&\, (1 << sh)$

The preprocessing of BAMs is presented in Alg. 11. The computation of the vectors I and F is similar to BAM, except that they allocate bins dynamically to each character, and the rest of the characters share the same bin. A counter $count$ calculates the number of occurrences of each character in the pattern and is used to allocate $\lfloor \log C_c \rfloor + 2$ bits dynamically for character c of the pattern, where C_c is the total count of character c in the pattern. In the preprocessing phase, a bit mask M_c is computed for each distinct character c that appears in the pattern. This bit mask is used to update the state vector D in the search phase. The bit mask I holds the initial values of the fields for each distinct character in the pattern and the bit mask F holds one bit at each overflow character. All these vectors, except state vector D are initialized in the preprocessing phase.

Let us consider an example having a pattern $P = abcdefggh$ and a text $T = abcdegfghx$ and the width of the computer word $w = 16$. As the length of the pattern is 8, the total number of bits required for all characters is 18. Since, the leftmost bit is reserved for the character that appears in the text window but not present in the pattern, there are 15 bits to store the characters of the pattern. It signifies that two characters of the pattern will share one bit. The initialization of the vectors I, M, D and F is shown below.

	x	h / g		f		e		d		c		b		a	
I	**0**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
D	**0**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
F	**1**	**1**	0	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0

	x	h / g		f		e		d		c		b		a	
M_a	**0**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	1
M_b	**0**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	1	**0**	0
M_c	**0**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	1	**0**	0	**0**	0
M_d	**0**	**0**	0	0	**0**	0	**0**	0	**0**	1	**0**	0	**0**	0	**0**	0
M_e	**0**	**0**	0	0	**0**	0	**0**	1	**0**	0	**0**	0	**0**	0	**0**	0
M_f	**0**	**0**	0	0	**0**	1	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
M_g	**0**	**0**	0	1	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
M_h	**0**	**0**	0	1	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
M_x	**1**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0

The rightmost character of the first text window $[abcdegfgh]x$ is processed and the vector D is updated by $D + M_h$ as shown below. The new value

of the vector D is tested against the vector F for finding any overflow. As no overflow bit is set in the vector D, the next character is processed from the current window $[abcdegf\underline{g}h]x$ at the position t_7. The value of the vector D is updated by he operation $D + M_g$, as shown below.

	x	h / g		f	e	d	c	b	a
D	**0**	**0** 0	0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0
M_h	**0**	**0** 0	1	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0
$D + M_h$	**0**	**0** 0	1	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0

	x	h / g		f	e	d	c	b	a
D	**0**	**0** 0	1	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0
M_g	**0**	**0** 0	1	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0
$D + M_g$	**0**	**0** 1	0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0

In the next step, the vector D is tested against the vector F, which results in false. The next character of the window is processed at position t_6 ($[abcdeg\underline{f}gh]x$). The update step is shown below by adding M_f and D. The whole text window is processed without setting any overflow bit and the execution reaches the leftmost character of the text window at position t_0. State vector D is updated by $D + M_a$. The test $D \ \& \ F$ is false and one permuted occurrence of the pattern in the text is reported.

	x	h / g		f	e	d	c	b	a
D	**0**	**0** 1	0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0
M_f	**0**	**0** 0	0	**0** 1	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0
$D + M_f$	**0**	**0** 1	0	**0** 1	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0

	x	h / g		f	e	d	c	b	a
D	**0**	**0** 1	1	**0** 1	**0** 1	**0** 1	**0** 1	**0** 1	**0** 0
M_a	**0**	**0** 0	0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 1
$D + M_a$	**0**	**0** 1	1	**0** 1	**0** 1	**0** 1	**0** 1	**0** 1	**0** 1

	x	h / g		f	e	d	c	b	a
D	**0**	**0** 1	1	**0** 1	**0** 1	**0** 1	**0** 1	**0** 1	**0** 1
F	**1**	**1** 0	0	**1** 0	**1** 0	**1** 0	**1** 0	**1** 0	**1** 0
$D \ \& \ F$	**0**	**0** 0	0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0	**0** 0

The text window is moved forward by one position. The new text window to be processed is $a[bcdegfghx]$. The rightmost character x of the current window is processed at position t_9. The vector D is updated by the

operation $D + M_x$ and the test of $D \ \& \ F$ turns out to be false, as shown below. The window is moved forward after the mismatch character (x). In the same manner, the whole text is processed and the total number of permuted occurrences of the pattern in the text is reported at the end.

	x	h / g		f		e		d		c		b		a	
D	**0**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
M_x	**1**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
$D + M_x$	**1**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0

	x	h / g		f		e		d		c		b		a	
D	**1**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0
F	**1**	**1**	0	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0	**1**	0
$D \ \& \ F$	**1**	**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**	0

5.3 BAM2

The BAM2 algorithm is a 2-gram variation of the BAM algorithm using the shared bin technique. As the algorithm follows a 2-gram approach, it handles two characters at a time. BAM2 has separate loops for patterns of even and odd lengths.

The loop for patterns of odd length has two extra execution lines because the remaining leftmost character of the alignment window must be handled in a different manner. The algorithm works by scanning the text window from right to left and processing two characters at a time. BAM2 adds the occurrence vector B and the state vector D for every two characters inside the text window and tests the state vector D against the overflow vector F for any overflow condition.

Typically q-grams are used in string matching to process the right end of the alignment window. BAM2 processes the whole window with 2-grams (except the leftmost character in the case of odd m). This is beneficial because the alignment window is scanned on average further to the left in jumbled matching than in ordinary string matching, since in jumbled matching we do not care about the order of characters. Moreover, 2-grams instead of single characters are read in our implementation of BAM2.

Algorithm 12 BAM2($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}$)

```
   /* Preprocessing */
 1: for all c ∈ Σ do M_c ← 0, C_c ← 0; sh ← 0
 2: for i ← 0 to m − 1 do
 3:       C_{p_i} ← C_{p_i} + 1
 4: for all i ∈ Σ do
 5:       if C_i = 0 then
 6:             M_i ← M_i & (1 << sh)
 7: for all i ∈ Σ do
 8:       for all j ∈ Σ do
 9:             B_{i,j} ← M_i + M_j
   /* Searching */
10: s ← 0
11: if m mod 2 = 0 then
12:       while s ≤ n − m do
13:             j ← s + m − 3; D ← I + B_{t+j−1,t+j}
14:             repeat
15:                   D ← D + B_{t+j−1,t+j}
16:                   if D & F then
17:                         break
18:                   j ← j − 2
19:             until j ≥ s
20:             if j < s then
21:                   Verify candidate match
22:                   s ← s + 1
23:             else
24:                   s ← j + 1
25: else
26:       while s ≤ n − m do
27:             j ← s + m − 3; D ← D + B_{t+j−1,t+j}
28:             repeat
29:                   D ← D + B_{t+j−1,t+j}
30:                   if D & F then
31:                         break
32:                   j ← j − 2
33:             until j ≥ s
34:             if j ≤ s then
35:                   D ← D + M_{t_j}
36:                   if D & F = 0 then
37:                         Verify candidate match
38:                         s ← s + 1
39:                   else
40:                         s ← j + 1
41: return occ
```

The pseudocode of BAM2 is presented in Alg. 12. In the preprocessing phase, the values of the initial vector I and occurrence vector F are calculated in a similar way as in BAM and BAMs. Array B is precomputed as follows: $B_{c_1,c_2} = M_{c_1} + M_{c_2}$. As two characters are handled at a time in BAM2, an additional vector B is used that holds the occurrences for

all possible two character combinations in the pattern. These values of the vector B are used to update the state vector D in the search phase by reading the characters t_{j-1} and t_j in a single instruction. In BAM2, B is computed for combination every two characters in the pattern, whereas in BAM, vector M is computed for every single character in the pattern.

In the search phase, BAM2 reads four characters before testing the vector D. As a consequence, the minimum width of a bit field is four bits instead of two. The width of the field for the characters not present in P is three bits. For large alphabets, we use BAM2 with the same bin sharing technique as applied in BAMs.

Let us consider an example having a pattern $P = acbccb$ and a text $T = acbcbbcacbc$ over an alphabet $\Sigma = \{a, b, c\}$. In this example, we are considering the dynamic allocation of the bins for each character. The initial values of the vectors I, M, B are represented as follows. The bin x is reserved for characters not in Σ. At the start of the execution, the first window of the text $[acbcbb]cacbc$ is processed.

	x		c			b			a	
M_a	**0**	**0**	0	0	**0**	0	0	**0**	1
M_b	**0**	**0**	0	0	**0**	0	1	**0**	0
M_c	**0**	**0**	0	1	**0**	0	0	**0**	0
M_x	**1**	**0**	0	0	**0**	0	0	**0**	0

	x		c			b			a	
B_{ab}	**0**	**0**	0	0	**0**	0	1	**0**	1
B_{ba}	**0**	**0**	0	0	**0**	0	1	**0**	1
B_{bc}	**0**	**0**	0	1	**0**	0	1	**0**	0
B_{cb}	**0**	**0**	0	1	**0**	0	1	**0**	0
B_{ac}	**0**	**0**	0	1	**0**	0	0	**0**	1
B_{ca}	**0**	**0**	0	1	**0**	0	0	**0**	1
B_{aa}	**0**	**0**	0	0	**0**	0	0	**1**	0
B_{bb}	**0**	**0**	0	0	**0**	1	0	**0**	0
B_{cc}	**0**	**0**	1	0	**0**	0	0	**0**	0

	x		c			b			a	
I	**0**	**0**	0	0	**0**	0	1	**0**	0
F	**1**	**1**	0	0	**1**	0	0	**1**	0
D	**0**	**0**	0	0	**0**	0	1	**0**	0

The value of the state vector D is updated by incrementing its value by the vector B_{bb} for the two rightmost characters of the text window $([acbc\underline{bb}]cacbc)$ at position t_5 and t_4 respectively. After addition, the value of the vector D is shown below.

	x		c			b			a	
D	**0**	**0**	0	0	**0**	0	1	**0**	0
B_{bb}	**0**	**0**	0	0	**0**	1	0	**0**	0
D	**0**	**0**	0	0	**0**	1	1	**0**	0

In the later step, the two characters at position t_3 and t_2 of the text window are accessed ($[ac\underline{bc}bb]cacbc$). The occurrence vector for the underlined characters i.e. B_{bc} is added to vector D. The updated value of vector D is shown below.

	x		c			b			a	
D	**0**	**0**	0	0	**0**	1	1	**0**	0
B_{bc}	**0**	**0**	0	1	**0**	0	1	**0**	0
D	**0**	**0**	0	1	**1**	0	0	**0**	0

The overflow bit of the state vector D corresponding to the character b at position t_2 is set as there are more than two occurrences of the character b in the text window. The test of vector D with the overflow vector F holds true, as shown below.

	x		c			b			a	
D	**0**	**0**	0	1	**1**	0	0	**0**	0
F	**1**	**1**	0	0	**1**	0	0	**1**	0
D & F	**0**	**0**	0	0	**1**	0	0	**0**	0

The text window is shifted over the character where an overflow occurred. The new text window is $acb[cbbcac]bc$. The value of state vector D is initialized and is updated again by adding the value of vector B_{ac} to it. The new value of state vector D is represented as follows. In the following step, the value of the state vector D is updated for the next two characters in the window ($acb[cb\underline{bc}ac]bc$) by adding the value of occurrence vector B_{bc} to it.

	x		c			b			a	
D	**0**	**0**	0	0	**0**	0	1	**0**	0
B_{ac}	**0**	**0**	0	1	**0**	0	0	**0**	1
D	**0**	**0**	0	1	**0**	0	1	**0**	1

	x		c			b			a	
D	**0**	**0**	0	1	**0**	0	1	**0**	1
B_{bc}	**0**	**0**	0	1	**0**	0	1	**0**	0
D	**0**	**0**	1	0	**0**	1	0	**0**	1

No overflow is reported after the addition and the execution scans the last two leftmost characters of the text window ($acb[\underline{cb}bcac]bc$). Thereafter, the value of the state vector D is updated by adding the value of B_{cb} to it. No overflow is detected after this addition, as shown below.

	x		c			b			a	
D	**0**	**0**	1	0	**0**	1	0	**0**	1
B_{cb}	**0**	**0**	0	1	**0**	0	1	**0**	0
D	**0**	**0**	1	1	**0**	1	1	**0**	1

The number of occurrences of the characters in the text window are the same as in the pattern, therefore no overflow bit is set as shown above. The final value of the vector D after the operation of $D \ \& \ F$ is shown below. The test $D \ \& \ F$ is true and one occurrence of the permuted pattern is reported. The text window is moved forward by one position. Pursuing in the same way, the algorithm scans the whole text and every candidate match must be verified. At the end of the execution, the total number of occurrences of the jumbled pattern in the text is reported.

	x		c			b			a	
D	**0**	**0**	1	1	**0**	1	1	**0**	1
F	**1**	**1**	0	0	**1**	0	0	**1**	0
$D \ \& \ F$	**0**	**0**	0	0	**0**	0	0	**0**	0

Analysis

We assume that the bit vector D fits to the computer word. The time complexity of BAM2 depends on two separate while loops for even and odd length patterns. In the worst case, the complexity is $O(mn)$. In the best case, the execution does not enter the inner conditional statements, resulting in the complexity $O(\frac{n}{m})$.

5.4 EBL

We present another algorithm for exact jumbled matching, EBL (Exact Backward for Large alphabets). EBL works for the large alphabets. It is

based on SBNDM2 [19], which is a sublinear bit-parallel algorithm for exact string matching. In BNDM, each alignment window is processed from right to left as in the Boyer–Moore algorithm [7] by simulating the nondeterministic automaton of the reversed pattern with bitparallelism. SBNDM2 starts processing each alignment window by reading a 2-gram.

EBL can be compared to BAM. In BAM, an increment vector (M_c) is used for each character c in the pattern, but in EBL, an array B states whether the character c is present in the pattern or not (B_c = 1, if c is present, otherwise B_c = 0). Also, the last two characters of the alignment window are processed before processing the remaining characters of the window. However, one character is processed at a time in BAM.

Algorithm 13 EBL($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do A_c ← 0
 2: for all c ∈ Σ do B_c ← 0
 3: for j ← 0 to m − 1 do
 4:       A_{p_j} ← A_{p_j} + 1
 5:       B_{p_j} ← 1
    /* Searching */
 6: i ← 0
 7: occ ← 0
 8: while i ≤ n − m do
 9:       j ← m − 2
10:       D ← B_{t_{i+j}} & B_{t_{i+j+1}}
11:       while D ≠ 0 and j > 0 do
12:             D ← D & B_{t_{i+j−1}}
13:             j ← j − 1
14:       if D = 1 then
15:             count ← −m
16:             for j ← 0 to m − 1 do
17:                   x ← p_j
18:                   C_x ← A_x
19:             for v ← 0 to m − 1 do
20:                   C_{t_{i+v}} ← C_{t_{i+v}} − 1
21:                   if C_{t_{i+v}} > 0 then
22:                         count ← count + 1
23:             if count ≥ 0 then
24:                   occ ← occ + 1
25:       i ← i + j + 1
26: return occ
```

In EBL, the element B_c corresponds to the character c and states whether the character c is present ($B_c = 1$) in the pattern or not ($B_c = 0$). As in SBNDM2, two characters are read before the first test in an alignment window. The update step of the state variable D is simply $D = D \,\&\, B_{t_{i+j+1}}$. When the alignment window contains only acceptable

characters, the window is a match candidate, which should be verified. Whenever a forbidden text character is found, the alignment window is moved forward over that text position.

The pseudocode of EBL is presented in Alg. 13. In the preprocessing phase, the frequency vector A and the occurrence vector B are initialized to zero. The initialization of the array C is not necessary. The frequency vector A and the occurrence vector B are updated according to the characters present in the pattern.

In the search phase, the text window is scanned in the backward direction. Each time, the last two characters of the current window are processed before processing the rest of the characters and the vector D is updated. The value of the vector $D = 0$, shows that the character is not present in the pattern and the value of counter i gets updated in line 25 and the window is shifted after the mismatch character of the text. But if the value of D is 1, the potential match is verified.

In the verification step in line 15, a variable $count$ is initialized by $-m$. The character count of each character present in the current text window is matched against the character count of each character in the pattern and the value of $count$ is updated. If the number of occurrences of each character in the pattern and the text window is same, an occurrence of the permuted pattern is reported in line 24 of the algorithm.

Let us consider an example having a pattern $P = abba$ and a text $T = bcabbacb$ over an alphabet $\Sigma = \{a, b, c\}$. As this is a backward approach algorithm, the text window is processed from right to left, skipping a significant portion of the text. In the preprocessing phase, the array A gets initialized to the number of occurrence of each character in the pattern. Hence, the value of $A_a = 2$ and $A_b = 2$.

The occurrence vector B is initialized to 1 for the characters that are present in the pattern. The value of vector B is $B_a = 1$, $B_b = 1$ and 0 for the rest of the characters, as they are not present in the pattern. In the searching phase, in line 10, state vector D is updated by the occurrence vector B to 1 ($bc\underline{a}\underline{b}bacb$) as a and b on location t_2 and t_3 are present in the occurrence vector B. Now the execution enters the $while$ loop in line 11. Vector D is updated until the condition is false.

The $while$ loop exits as the execution scans character c at location t_1 of the text, since $B_c = 0$ for c. The text window is shifted to the location t_2 ($bc[abba]cb$). Likewise, the vector D is updated in step 10 of the algorithm.

Once again, the execution enters the $while$ loop in line 11, as both the conditions hold true. After verifying the condition of $while$ loop, the execution enters the conditional if statement since the value of D is one. In the lines from 14 to 23 of the algorithm, the candidate match is verified. The verified match is reported and the total number of occurrences of the permuted pattern is reported at the end of the algorithm.

Analysis

Algorithm EBL is based on SBNDM2, which is a sublinear bitparallel algorithm for exact string matching. The main loop of the algorithm is $while$ loop in line 8. In the best case, the algorithm works sublinearly (for small alphabets) by skipping a significant portion of the text. In the worst case, the complexity is $O(nm)$ as inner for loops in line 16 and 19 execute m times and outer $while$ loop in line 8 executes n times.

5.5 EFS

In this section, we present the EFS (Exact Forward for Small alphabets) algorithm for exact jumbled matching. EFS works for small alphabets. It applies counters of BAM to forward scanning. Like in BAM, D is tested with a mask F, which has one at each overflow bit. The vector M_c is a bit mask having one at the least significant bit of the field of character c. In this way, EFS is a cross of BAM and Count. It uses the vectors D, F and M of BAM.

An occurrence vector M_c holds one set bit corresponding to the occurrence of character c in the pattern. It updates state vector D for each character in the text window by a simple step $D = D + M_c$. The state vector D has a field of d bits, initially $2^{d-1} - cc(P, c) - 1$ for each character c appearing in P. The characters not in P have a joint field of one bit. An overflow vector F is used to test the overflow bits of the state vector D. The overflow bits corresponding to each character in the pattern are set as 1 in vector F.

The pseudocode of EFS is presented in Alg. 14 for 4 characters. It is assumed that the text does not contain other characters. In the preprocessing phase, the variables fc and d correspond to field count and field width respectively. The occurrence vector M_c, for each character in the character set Σ is initialized for by setting the least significant bit as 1

in each counter. The state vector D is initialized by setting all bits as 1, except the overflow bits corresponding to each character in the pattern. In the next step, the state vector D is updated as $D = D - M_c$, which creates a space for the characters to be added while processing the text window.

In the search phase, the first alignment window is processed before scanning the rest of the text. The state vector D is tested with overflow vector F, which has bit one at each overflow bit. Before testing the state vector D with the vector F, the vector M_c is added to the state vector D. If the result is true, then there is an occurrence of the permuted pattern in the text window and it is reported in line 15.

Algorithm 14 EFS($P = p_0p_2 \cdots p_{m-1}, T = t_0t_2 \cdots t_{n-1}$)

```
    /* Preprocessing */
 1: fc ← 4; d ← w/fc
 2: for all c ∈ Σ do M_c ← 1
 3: x ← 1 << (d * (fc − 1))
 4: y ← x
 5: for all i ∈ Σ do M_i ← x; x ← x >> d;  y ← y | x
 6: F ← y << (d − 1)
 7: D ← F − y
 8: for i ← 0 to m − 1 do
 9:       D ← D − M_{p_i}
    /* Searching */
10: occ ← 0
11: for i ← 0 to m − 1 do
12:       D ← D + M_{t_i}
13: while i < n do
14:       if D & F = 0 then
15:             occ ← occ + 1
16:       D ← D + M_{t_i} − M_{t_{i−m}}
17:       i ← i + 1
18: return occ
```

Let us consider an example having a pattern $P = accc$ and a text $T = aacaccgcgtga$ over an alphabet $\Sigma = \{a, c, g, t\}$ for $w = 24$ (we consider input as clean DNA). The array M of bitvectors stores the bits having one at least significant bit of the field for each character in the same manner as in previous algorithms. The value of mask F and the initial value of the state vector D are shown below.

	a						c						g						t					
F	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0
D	**0**	1	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1

As the pattern is $accc$, the bits corresponding to the characters a and c

are updated (making room for the bits to be added in the text window) and the rest of the bits remain same. There is one occurrence of character a and three occurrences of character c in the pattern, which results in the updated value of vector D as shown below.

	a					c						g						t					
D	**0**	1	1	1	1	0	**0**	1	1	1	0	0	**0**	1	1	1	1	1	**0**	1	1	1	1	1

In the searching phase, the first alignment window of the text $[aaca]ccgcgtga$ is processed before the main loop. The value of the vector M_a is added to vector D for the leftmost character read at position t_0 represented below.

	a					c						g						t					
D	**0**	1	1	1	1	0	**0**	1	1	1	0	0	**0**	1	1	1	1	1	**0**	1	1	1	1	1
M_a	**0**	0	0	0	0	1	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0
D	**0**	1	1	1	1	1	**0**	1	1	1	0	0	**0**	1	1	1	1	1	**0**	1	1	1	1	1

After processing the first alignment window, the value of vector D is shown as follows. As there is more than one occurrence of character a in the text window, the overflow bit corresponding to character a in state vector D is set indicating an overflow.

	a					c						g						t					
D	**1**	0	0	0	0	1	**0**	1	1	1	0	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1
F	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0
D & F	**1**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0

As a result, the test of vector D with vector F is false and the window is moved forward by one position $a[acac]cgcgtga$ without reporting any occurrence of pattern in the text. The execution continues by reading the characters at the positions t_0 and t_4. The state vector D is updated by subtracting the occurrence vector M_a and adding M_c to it. Thereafter, at the end of the current text window, no match is reported and the window is moved forward by one position ($aa[cacc]gcgtga$). In this window, the first occurrence of the permuted pattern is reported.

	a					c						g						t					
D	**0**	1	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	1	1
F	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0	**1**	0	0	0	0	0
D & F	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0

Pursuing the execution in this way, the whole text is scanned from left to right by updating the state vector D for each character and shifting the window by one position each time and reporting the total number of occurrences of the permuted pattern in the text at the end.

Analysis

The main loop of the algorithm is $while$ loop in line 13. The time complexity of EFS is $\Theta(n)$.

5.6 EFB

Algorithm 15 EFB($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}$)

```
   /* Preprocessing */
 1: X ← 0
 2: for i ← 0 to m − 1 do
 3:       X ← X + p_i − 48
   /* Searching */
 4: C ← 0; occ ← 0
 5: for j ← 0 to m − 1 do
 6:       C ← C + t_j − 48
 7: while j < n do
 8:       if C = X then
 9:             occ ← occ + 1
10:       C ← C − t_{j−m} + t_j
11:       j ← j + 1
12: return occ
```

EFB (Exact Forward for Binary) is a forward type exact algorithm that finds the permuted occurrences of the pattern in the text for binary data. It is a variation of the Count algorithm tuned for binary data containing only ASCII characters 0 and 1. The main idea of this algorithm is to count the number of 1's in the pattern and compare that count with the number of 1's in each text window. If the count is the same, an occurrence of the permuted pattern in the text is reported. The whole text is scanned by moving forward one position and updating the count of 1's in each text window and comparing it with the count of 1's in the pattern. The motivation for developing a specialized algorithm for binary data is that general algorithms give poor results.

The pseudocode of EFB is presented in Alg. 15. In the preprocessing phase, a variable X is used to count the total number of 1's in the pattern. The variable X is given a value 0 in the beginning. Since the ASCII value

of 0 and 1 is 48 and 49 respectively, we subtract all occurrences of 0's and 1's by 48, to count the number of 1's in the pattern. The total count of 1's in the pattern is stored in the variable X.

In the search phase, the variable C is used to count the number of 1's in the text window. Similar to the Count algorithm, the first text widow is processed before processing the rest of the text in EFB. In the main $while$ loop in line 6, the variables C and X are compared to find whether the total number of 1's are equal in the text window and the pattern. The text window is moved forward by one position. A new character is added and the leftmost character of the previous window is removed. The variable C is updated for the new window in line 10 of EFB and the variables C and X are compared. In a similar manner, the whole text is processed by moving forward one position and updating the value of variable C of the new window. The total number of occurrence of the permuted pattern in the text is reported at the end.

Let us consider an example having a pattern $P = 1011$ and a text $T = 001111000$ over the alphabet $\Sigma = \{0, 1\}$. The count of 1's in the pattern is 3. The first text window ($[0011]11000$) is processed and the count of 1's is 2. This count is compared against the count in the pattern. Since the count of 1's in the pattern is three, no occurrence of the permuted pattern is reported and the text window is moved forward by one position.

The count of 1's in the new text window ($0[0111]1000$) is computed which comes out to be 3. This count of 1's is compared against the count of 1's in the pattern. Since the count of 1's is the same in both the text window and the pattern, one occurrence of the permuted pattern is reported. The text window is moved forward by one position and the count of 1's is updated. The whole text is scanned in the same way and the total number of occurrences of the permuted pattern in the text (i.e. 2) is reported at the end.

Analysis

EFB algorithm is based on the Count algorithm. The main loop of EFB is in line 7. The algorithm scans the whole text by moving forward one position yielding the complexity of $\Theta(n)$.

6. New Methods for Exact JPM with SIMD

In this chapter, we introduce two algorithms for exact jumbled matching of short patterns. These algorithms apply SIMD (Single Instruction Multiple Data) computation in order to quickly filter the text. We developed several algorithms using SIMD techniques, but we are presenting only the best among them.

6.1 SIMD Instructions in Use

In our algorithms, we use specialized string matching SIMD instructions in addition to standard SIMD instructions. We also make use of aggregation operations that are included in SIMD architecture, to process more than one character simultaneously. In our approach, we have applied the equal any operation to speed up reading the text. The operation has two input operands which are strings of upto 16 characters. The first string represents a multiset of characters. The second string is the text itself. The output of the operation is a bitvector of 16 bits, where 1 means that the corresponding character in the string belongs to the set and 0 means the opposite. For instance, let us consider a string *abbc* representing the multiset $\{a, b, b, c\}$ and a string *adcdbeaeebefdbce* as operands. The output of the equal any operation is 1010101001000110 in the reverse order.

We have used $simd\text{-}load$, $simd\text{-}equal\text{-}any(x, y)$ and $simd\text{-}cmpeq(x, y)$ SIMD instructions in our algorithms. The instruction $simd\text{-}load$ is formally

$$\_\_m128i \ \_mm\_loadu\_si128 \ (\_\_m128i \ const* \ mem\_addr).$$

This load instructions for SSE memory operation loads 128 bits of integer data from the memory location.

The instruction, $simd\text{-}equal\text{-}any(x, y)$ is formally

$$\_mm\_extract\_epi16\ (\_mm\_cmpistrm\ (x, y, \text{\_SIDD\_CMP\_EQUAL\_ANY})).$$

The inner instruction contains three parameters. The first and second parameters are string fragments with a maximum size of 16 bytes. The third parameter is a constant, determining the type of comparison to be performed and the format of value to be returned. In our case, the third parameter is _SIDD_CMP_EQUAL_ANY. The instruction $\_mm\_extract\_epi16$ extracts a selected signed or unsigned 16-bit integer from two of its parameters.

The instruction $simd\text{-}cmpeq(x, y)$ is formally

$$\_mm\_movemask\_epi8\ (\_\_m128i\ \_mm\_cmpeq\_epi8\ (\_\_m128i\ x, \_\_m128i\ y)).$$

The instruction $\_mm\_movemask\_epi8\ (\_\_m128i\ z)$ creates a mask from the most significant bit of each 8-bit element in the parameter z and stores the result. The instruction $\_\_m128i\ \_mm\_cmpeq\_epi8\ (\_\_m128i\ x, \_\_m128i\ y)$ compares packed 8-bit integers in x and y for equality and stores the result.

6.2 Equal Any Approach

The algorithm in this section uses equal any SIMD command. Let us assume that $m < 16$ holds. The width of a test window in the text is 16. The equal any SIMD command returns a bitvector k of 16 bits showing the positions in the test window which hold any character of the pattern. For example, if the test window is *this is a sample* and the pattern is *aeiou*, then the vector is:

```
elpmas a si siht
1000100100100100
```

It is important to note that the orientation of the bitvector is the opposite of the text. A match candidate is found if the last m bits of the vector are ones. Note that such a case is only a match candidate because the counts of characters are not analyzed. For example, the string *aaaaa* is a match candidate for *abcde*.

The pseudocode of the scanning algorithm EA based on the equal any

approach is shown in Alg. 16. In EA, we use a table d for shifting the test window. The algorithm applies a skip loop with a stopper, which is a copy of the pattern. A heuristic algorithm called PP to compute d from m is given as Alg. 17. When a block of m ones is found in EA, the window is shifted so that the block is at the right end of the bitvector corresponding to the test window. When a block of m ones is at the right end, a match candidate is found. The entry of d is zero in such a case in order to get out from the skip loop.

Algorithm 16 EA($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}$)

```
 1: Place a copy of P after T
 2: Call PP(m)
 3: occ ← 0; x ← simd-load(P)
 4: shift ← 1; i ← 0
 5: while true do
 6:     while shift > 0 do
 7:         y ← simd-load(t_i ··· t_{i+15})
 8:         k ← simd-equal-any(x, y)
 9:         shift ← d_k
10:         i ← i + shift
11:     occ ← occ + verify(t_i ··· t_{i+m-1})
12:     if i > n − m then
13:         return occ − 1
14:     i ← i + 1
```

Now we cover the detailed explanation of EA. The SIMD registers x and y hold the pattern and a test window of 16 bytes of the text respectively. The registers x and y are processed with the $simd\text{-}equal\text{-}any$ operation resulting a 16-bit integer k on line 8. If $d_k = 0$ holds, a match candidate is found and the inner loop is exited. On line 11 there is a call of a verification routine which verifies the match candidate. Any previous algorithm for jumbled pattern matching (especially BAM, BAM2, or EBL) can be used as a verification method. The variable occ holds the count of matches. The stopper creates a superfluous match which is subtracted from the count on line 13.

In the algorithm PP, the shift table d is indexed with the 16-bit integer k. The computation consists of several subsequent for-loops which are in the decreasing order of shift. This order of computation is essential, because a single entry of d may be assigned several times.

Algorithm 17 PP(m)

```
 1: L ← 2^16 − 1; b ← 1 << 15
 2: for i ← 0 to b − 1 do
 3:     d_i ← 16
 4: a ← b; b ← 3 << 14
 5: for x ← 15 downto 16 − m do
 6:     for i ← a to b − 1 do
 7:         d_i ← x
 8:     a ← b; b ← b + (1 << (x − 2))
 9: s ← (1 << m) − 1; a ← s << (15 − m)
10: e ← 1 << 15; c ← 1 << (14 − m)
11: for x ← 15 − m downto 2 do
12:     for i ← a step b to L do
13:         for j ← 0 to c − 1 do
14:             d_{i+j} ← x
15:     a ← a >> 1; b ← b >> 1; c ← c >> 1
16: for i ← a step b to L do
17:     d_i ← 1
18: for i ← s step s + 1 to L do
19:     d_i ← 0
```

Let us go through the phases of Alg. PP in detail. Let the 16 bits of k be named by $k_{15}, k_{14}, \cdots, k_0$. In the beginning, the integer e corresponds to a bitvector of one followed by 15 zeros. When the leftmost bit k_{15} is zero, the test window can be shifted 16 positions in the best case (lines 1–3). On lines 4–8 we consider the case where k starts with $16 - x$ ones followed by a zero for $x = 15, 14, \cdots, 16 - m$. Then the shift is x. For example, the shift for 1111010101001100 is 12 for $m > 4$.

We consider the case where k holds a block of m ones starting from $k_{14}, k_{13}, \cdots$ or k_{m+1} on lines 9 – 15. The loop for x traverses all possible locations of the block of m ones. The loop for i traverses all bit combinations of the first $16 - x - m$ bits. The loop for j traverses all bit combinations of the last x bits. For example, in the computation of $d[0011111101000100]$ for $m = 6$, a is 0011111100000000, b is 0100000000000000, c is 0000000100000000 and x is 8.

When the block of m ones ends at k_1, the shift is one and it is computed in a single loop (lines 16–17). When the block of m ones ends at k_0, a match candidate is found, and this is expressed as assigning a zero to all such entries (lines 18–19).

Let us consider an example having a pattern $P = bbcca$ and a text $T = abcdabccbddbcadd$ over an alphabet $\Sigma = \{a, b, c, d\}$. The equal any relation operation will result into a string of 0011100111110111. As the length of pattern is 5, a match candidate must contain 5 consecutive ones.

At position t_4, there is an occurrence of 5 consecutive ones. When a block of 5 ones is found, the window is shifted so that the block is at the left end of the bitvector corresponding to the test window. One of our algorithms (such as EBL) can be used to verify the candidate match. After verification, one occurrence of the permuted pattern is reported. The window is moved forward by one position and a chunk of next 16 characters of the text is considered for equal any operation. The whole text is scanned the same way and the total number of permuted occurrences of the pattern in the text is reported at the end.

Analysis

In the worst case, the EA algorithm needs $O(nm)$ time. In the best case, the algorithm is linear, and the SIMD commands offer a practical speed-up, because only a few commands are executed for each block of 16 characters.

6.3 Least Frequent Character Approach

Our second approach was developed for natural language. We use SIMD instructions to analyze whether a test window of 16 bytes holds the least frequent character of the pattern. The frequency of characters is based on the text or on the language.

Algorithm 18 LF($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, R$)

```
1:  Place a copy of P at t_{n+1}
2:  occ ← 0; i ← 0; f ← 0
3:  x ← simd-load(R)
4:  while true do
5:      y ← simd-load(t_i ··· t_{i+15})
6:      while simd-cmpeq(x, y) = 0 do
7:          i ← i + 16
8:          y ← simd-load(t_i ··· t_{i+15})
9:      if i < n − m then
10:         f ← i − m + 1
11:     occ = occ + search(t_f ··· t_{i+15+m−1})
12:     f ← i + 16
13:     if f ≥ n then
14:         return occ − 1
15:     else
16:         i ← i + 16
```

The pseudocode of the scanning algorithm LF is shown in Alg. 18, which is based on the least frequent character approach. It is assumed that be-

fore execution, T is in a buffer followed by at least $2m+14$ null characters. On line 11 there is a call of a search routine which searches a block of up to $2m+14$ characters. Any previous algorithm for jumbled pattern matching (especially BAM, BAM2, or EBL) can be used as a search method. The parameter R is an array containing 16 bytes, each of which holds the least frequent character. The SIMD register x holds R and the SIMD register y holds a test window of 16 bytes of the text. The registers x and y are compared by the $simd\text{-}cmpeq$ operation on line 6. The algorithm applies a skip loop with a stopper, which is a copy of the pattern. As in Alg. EA, the stopper creates a superfluous match which is subtracted from the count on line 14.

An additional variable f is used to control the starting position of a block. If the previous block has been skipped, then the leftmost possible starting position for a match is $i+m-1$. Otherwise, the leftmost possible starting position for a match is i. Without such control, we would get the wrong number of matches, because there could be matches that would belong to two blocks.

Let us consider an example, having text $T = ILGLIENYAKIIAYK$ and pattern $P = YAK$. Here, T and P are taken from protein data. As this algorithm is based on the least frequent character of the text in the pattern, array R now contains character Y at each byte. The SIMD register x holds R and the SIMD register y holds a test window of 16 bytes of the text. Then, the registers x and y are compared by the $simd\text{-}cmpeq$ operation. As a result it is determined that the least frequent character Y is present in the pattern. Using this location in the text, one of our algorithms (such as BAM2) can be used as checking routine to find out the actual occurrence of the permuted pattern in the text window. As pattern YAK is present in the text, two occurrences of the pattern in the current text window are reported.

Thereafter, the scanning is moved forward according to the next position of character Y in the text window i.e at the position t_{16}. Here, another occurrence of the pattern is found. This is the stopper occurrence which is subtracted from the count. In this manner, the whole text is scanned and the total number of occurrences of the permuted pattern in the text is reported at the end.

Analysis

In the worst case, the LF algorithm works linearly, if the subroutine *search* is linear. In the best case, the algorithm is also linear, but the SIMD commands offer a practical speedup.

7. New Methods for Approximate JPM

In this chapter, we present new algorithms for approximate jumbled matching under the substitution model. All algorithms work by the sliding window approach. The first algorithm is ABAM which is an approximate variation of the BAM algorithm. ABAM works in a similar manner as BAM and it also finds the approximate occurrences of jumbled patterns. The second algorithm is AF, which is a modification of Mcount presented in Chapter 4.

Furthermore, we present the ABS algorithm, which scans the text window in the backward direction and works for small alphabets. The fourth algorithm is ABL, which is a variation of ABS, working for larger alphabets. It works for the patterns having characters whose bins do not fit the computer word, allowing one of the bins to be shared.

Yet we present another algorithm AJSM, which uses a two-dimensional array of integers (zeros and ones) to represent the counts of characters in the pattern. The next algorithm is AJMRS, which also uses the bitvectors to represent the counts of characters in the pattern. The central idea of this algorithm is to use the left and right rotations to update the bitvector. The last algorithm we present is AFB, which is an approximate version of EFB for binary data.

7.1 ABAM

ABAM (Approximate Bit-parallel Abelian Matching) is an approximate version of BAM. In ABAM, F_c is a bitvector having one at the overflow bit of the counter field for the character c. In BAM, the vector F is shared for all characters, but in ABAM, vector F_c is computed separately for every character c. State vector D holds the counters for the characters. A variable C is used for counting the errors. ABAM can have dynamic field

widths like DBAM, but for simplicity, here we present ABAM with a constant field width. The width of the field for a character in the pattern is $\lfloor log\ m \rfloor + 2$. For the characters that are not present in P, the width of the field is $\lfloor \log\ k \rfloor + 2$. The vectors M_c and I work in the same manner as in the BAM algorithm. The vector I holds the initial values for each counter and the bit mask M_c is used to update the state vector D when the character c is processed.

Algorithm 19 ABAM($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Searching */
 1: s ← 0; C ← 0; occ ← 0
 2: while s ≤ n − m do
 3:     D ← I; j ← s + m − 1
 4:     while j ≥ s do
 5:         D ← D + M_{t_j}
 6:         C ← C + (if D & F_{t_j} ≠ 0 then 1 else 0)
 7:         if C > k then
 8:             break
 9:         j ← j − 1
10:         if j < s then
11:             occ ← occ + 1
12:             s ← s + 1
13:         else
14:             s ← j + 1
15: return occ
```

The searching phase of ABAM is presented in Alg. 19. The preprocessing phase of ABAM is similar to BAM, except for the calculation of vector F. Vector F_c has one at the overflow bit of the counter field for the character c. For each character x not present in the pattern, the vector F_x is calculated respectively. Every bit of the counter field is one in F_x.

In the search phase, ABAM works by initializing state vector D with initial vector I at each window. As ABAM scans the text window from right to left, it updates the vector D by adding the occurrence vector M_c for character c. A mismatch counter C is used to keep track of the number of mismatches in the current window. If the number of mismatches is greater than the error count k, then the text window is moved forward after the overflow character. The total number of permuted occurrences of the pattern in the text is reported at the end.

Let us consider an example with a pattern $P = abbcb$ and a text $T = cabbaacba$ over an alphabet $\Sigma = \{a, b, c, d\}$ for error value $k = 1$. This algorithm works by processing the text window in the backward direction. The values of the vectors I, F_c and M_c are calculated for each character c

in the preprocessing phase and are represented as follows. The first alignment window is $[cabba]acba$.

	x		c				b				a			
M_a	**0**	**0**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	1
M_b	**0**	**0**	**0**	0	0	0	**0**	0	0	1	**0**	0	0	0
M_c	**0**	**0**	**0**	0	0	1	**0**	0	0	0	**0**	0	0	0
M_x	**0**	**1**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	0

	x		c				b				a			
I	**0**	**0**	**0**	1	1	0	**0**	1	0	0	**0**	1	1	0
D	**0**	**0**	**0**	1	1	0	**0**	1	0	0	**0**	1	1	0

	x		c				b				a			
F_a	**0**	**0**	**0**	0	0	0	**0**	0	0	0	**1**	0	0	0
F_b	**0**	**0**	**0**	0	0	0	**1**	0	0	0	**0**	0	0	0
F_c	**0**	**0**	**1**	0	0	0	**0**	0	0	0	**0**	0	0	0
F_x	**1**	**1**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	0

In the searching phase, state vector D is initialized by vector I as shown above. The first window ($[cddba]acba$) of the text is processed. The algorithm reads the last character of the window at position t_4 and updates the state vector D by adding the value of vector M_a.

	x		c				b				a			
D	**0**	**0**	**0**	1	1	0	**0**	1	0	0	**0**	1	1	1

The variable C for the number of mismatches is updated in the next step, i.e., with the value 0 without breaking the loop. The execution reads the next character at position t_3. After reading the character b ($[cdd\underline{b}a]acba$), the value of the vector D is updated as shown below.

	x		c				b				a			
D	**0**	**0**	**0**	1	1	0	**0**	1	0	1	**0**	1	1	1

In the next step, the character d at position t_2 is read and the value of vector D is updated by adding the value of occurrence vector M_x for the character not present in the pattern. The state vector D is tested against the overflow vector F. The test result is true which leads to the increment of mismatch counter C by 1. As the value of the variable C is not greater than the error value k, the execution reaches at the next character of the text window, i.e., character d at position t_1.

	x		c				b				a			
D	**0**	**1**	**0**	1	1	0	**0**	1	0	1	**0**	1	1	1
F_x	**1**	**1**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	0
D & F_x	**0**	**1**	**0**	0	0	0	**0**	0	0	0	**0**	0	0	0

Since there is an overflow for the character d, the state vector D and mismatch counter are updated. Now the value of C is 2, which is greater than the error value k, therefore the execution shifts the window after the overflow character at position t_2.

The new window to be processed is $cd[dbaac]ba$. Again the state vector D is initialized by the vector I and the window is processed in the backward manner. In a similar way, the whole text is scanned by shifting the text window and the total number of occurrences of the permuted pattern in the text is reported at the end.

Analysis

There are two main loops of the algorithm ABAM in the search phase. The worst case complexity of the algorithm is $O(nm)$, as each time the inner while loop executes m times in the worst case. In the best case, the algorithm works sublinearly, yielding the complexity of $O(\frac{kn}{m-k})$.

7.2 AF

We present another approximate algorithm AF (Approximate Forward). It is a modification of Mcount tuned for a single pattern.

The array E of bitvectors as well as the offset l are the same as in Mcount for a single pattern. The array E holds the counters of characters and the offset l is used to move the overflow bit of the corresponding counter to the right end. The initial value of the error counter C is $-(m-k)$. The occurrence of a permuted pattern is reported, if the value of C is greater or equal to zero.

The pseudocode of AF is presented in Alg. 20. In the preprocessing phase, the offset l is given the value $\lceil log\ m \rceil$.

Algorithm 20 AF($P = p_0p_1 \cdots p_m, T = t_0t_1 \cdots t_{n-1}, k$)

/* Preprocessing */
1: $l \leftarrow \lceil log_2\ m \rceil$
2: **for all** $c \in \Sigma$ **do** $E_c \leftarrow\sim 0 >> w - l$
3: **for** $j \leftarrow 0$ **to** $m - 1$ **do**
4: $\quad E_{p_j} \leftarrow E_{p_j} + 1$
/* Searching */
5: $j \leftarrow 0; occ \leftarrow 0$
6: **while** $j < m$ **do**
7: $\quad c \leftarrow t_{j+1}$
8: $\quad C \leftarrow C + (E_c >> l)$
9: $\quad E_c \leftarrow E_c - 1$
10: $\quad j \leftarrow j + 1$
11: **while** $j < n$ **do**
12: $\quad$ **if** $C \geq 0$ **then**
13: $\quad\quad occ \leftarrow occ + 1$
14: $\quad E_{t_{j-m}} \leftarrow E_{t_{j-m}} + 1$
15: $\quad C \leftarrow C + (E_{t_j} >> l) - (E_{t_{j-m}} >> l)$
16: $\quad E_{t_j} \leftarrow E_{t_j} - 1$
17: $\quad j \leftarrow j + 1$
18: **return** occ

In the search phase, the first text window is processed before processing the remaining text. The vector E_c is updated for each character c in this text window. After the first window has been processed, the value of mismatch counter C is updated as $C + (E_{t_j} >> l) - (E_{t_{j-m}} >> l)$ for each position j. There are two operations to update C in Mcount and only one operation in AF. This leads to the faster execution of AF. For each window, the value of C is tested. If there is success, then an occurrence of pattern in the text is reported and the window is moved forward by one step.

Let us consider an example having a pattern $P = aaabbc$ and a text $T = aadbcbbbabbcdab$ with the value of $k = 2$ over an alphabet $\Sigma = \{a, b, c, d\}$. The value of offset l is 3. The value of mismatch counter C is initialized with a value $-(m - k)$. The initial values of E are shown below, before processing the text, where E_x corresponds to the value for the character x that is not present in the pattern.

E_a	1	0	1	0
E_b	1	0	0	1
E_c	1	0	0	0
E_x	0	1	1	1

The leftmost text window, $[aadbcb]bbabbcdab$ of T is processed first. The values of array E are updated after processing each character of the first

window. The updated values of E_c for each character c in the first text window and are shown as follows.

E_a	1	0	0	0
E_b	0	1	1	1
E_c	0	1	1	1
E_x	0	1	1	0

After processing the last character of the first window the mismatch count C is tested, whether its value is greater than or equal to 0. The test is true and one match is reported in the first window. The window is moved forward by one step and the next window to be processed is $a[adbcbbb]abbcdab$. In the same way, the mismatch counter C is updated and the array E is updated for the new character of the text window and each match in the current window is reported. The whole text is scanned in the same manner and the total number of occurrences of the permuted pattern in the text is reported at the end.

Analysis

The time complexity of AF algorithm is $O(m + \Sigma)$ in the preprocessing phase. In the search phase, AF works linearly, having the time complexity of $O(n)$, since there are two successive main loops in lines 6 and 11 taking $O(m)$ and $O(n - m)$ time respectively.

7.3 ABS

The ABS (Approximate Backward Small alphabets) algorithm works for small alphabets. We have tested this algorithm on DNA data. The pseudocode of ABS is shown in Alg. 21. The state vector E holding the counters (or bins) of characters is initialized for each alignment window and has initially $2^{d-1} - cc(P, c) - 1$ (in the field of d bits) for each character c appearing in P and a joint field for characters not in P. The offset d_c is used to move the overflow bit of a field to the right end. B_c is a bit mask having one at the least significant bit of the field of character c. Before processing each alignment window, the state vector E is initialized for the counters of characters.

Algorithm 21 ABS($P = p_0p_1 \cdots p_m, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Searching */
 1: i ← m − 1
 2: occ ← 0
 3: while i < n do
 4:     C ← 0; E ← I
 5:     while (C ≤ k) & (i > i − m) do
 6:         E ← E + B_{t_i}
 7:         C ← C + ((E >> d_{t_i}) & 1)
 8:         i ← i − 1
 9:     if C ≤ k then
10:         occ ← occ + 1
11:     i ← i + m + 1
12: return occ
```

Let us consider an example having a pattern $P = ggaccg$ and a text $T = actgtagggccgcata$ over an alphabet $\Sigma = \{a, c, g, t\}$ for $k = 0$. The algorithm works by backward scanning of the text window. At the beginning, the leftmost window is $[actgta]gggccgcata$. The value of state vector E and occurrence vector B_c for each character c is calculated in the preprocessing phase. The value of the state vector and occurrence vector for each character is presented as follows.

	x						a						c						g						t					
B_a	**0**	0	0	0	0	0	**0**	0	0	0	0	1	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0
B_c	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	1	**0**	0	0	0	0	0	**0**	0	0	0	0	0
B_g	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	1	**0**	0	0	0	0	0
B_t	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	1
B_x	**0**	0	0	0	0	1	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0

	x						a						c						g						t					
E	**0**	1	1	1	1	1	**0**	1	1	1	1	0	**0**	1	1	1	0	1	**0**	1	1	1	0	0	**0**	1	1	1	1	1

The processing of the current text window begins by adding the occurrence vector of the rightmost character at position t_5 of the present window, i.e., $[actgt\underline{a}]gggccgcata$ to the state vector. One occurrence of character a does not set the overflow bit, as represented in the following table. The number of mismatches is zero, so far.

	x						a						c						g						t					
E	**0**	1	1	1	1	1	**0**	1	1	1	1	0	**0**	1	1	1	0	1	**0**	1	1	1	0	0	**0**	1	1	1	1	1
B_a	**0**	0	0	0	0	0	**0**	0	0	0	0	1	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0
$E + B_a$	**0**	1	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	0	1	**0**	1	1	1	0	0	**0**	1	1	1	1	1

The next character at position t_4 is scanned $T = [actg\underline{t}a]gggccgcata$. The character t is not present in the pattern, hence the overflow bit for the rest

of the characters is set as shown below. Now, there is one mismatch and the mismatch counter C is incremented by 1 and no match is reported.

	x						a						c						g						t					
E	**0**	1	1	1	1	1	**0**	1	1	1	1	1	**0**	1	1	1	0	1	**0**	1	1	1	0	0	**0**	1	1	1	1	1
B_x	**0**	0	0	0	0	1	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0	**0**	0	0	0	0	0
$E + B_x$	**1**	0	0	0	0	0	**0**	1	1	1	1	1	**0**	1	1	1	0	1	**0**	1	1	1	0	0	**0**	1	1	1	1	1

The window is moved forward by four positions at t_5. Again, the state vector E is initialized and the mismatch counter is initialized to zero. The new window to be processed is $T = actgt[agggcc]gcata$. Now the next character to be processed is c at position t_{10}. The whole window is scanned without setting any overflow character in the state vector and no mismatch occurred. Therefore, a match in the current window is reported. The window is moved forward by one step. The whole text is scanned by shifting the window forward and reporting all occurrences of the pattern in the text.

Analysis

The ABS algorithm is sublinear in the best case. In the worst case, the innermost $while$ loop executes m times, yielding $O(mn)$ total time.

7.4 ABL

The ABL (Approximate Backward Large alphabets) algorithm is a variation of the ABS algorithm for larger alphabets. It uses the approach of shared bins, which means that if there are not enough bins for all characters of the pattern, the rest of the characters share a bin. Then instead of reporting an occurrence of the pattern in the text, each candidate match needs to be verified. The preprocessing phase of the ABL algorithm is straightforward and is similar to the preprocessing of BAMs and ABS.

The ABL algorithm works by associating a counter with each character in the pattern. If the pattern is long, w (the number of bits in the computer word) bits are not enough to hold a distinct bin for each character appearing in the pattern. Some of the characters share the last fitting bin. State vector E and occurrence vector B_c for character c are processed in a similar way as in ABS. Variable C is used to count the number of mismatches in the current text window. If the number of mismatches is less than or equal to the error value k, then a candidate match is found and instead

of reporting an occurrence (line 10 in ABS), the candidate match needs to be verified for the actual occurrence of the permuted pattern in the text. At the end, the total number of occurrences of the permuted pattern is reported.

Analysis

As the ABL algorithm is a variation of the ABS algorithm for larger alphabets, it works sublinearly in the best case. In the worst case, the algorithm takes $O(nm)$ time.

7.5 AJMS

AJMS (Approximate Jumbled Matching by Shifts) [22] is a forward type algorithm that works for approximate jumbled matching. The algorithm executes by sliding a window of size m over the text. In the beginning, the left end of the text window and the pattern are aligned.

The algorithm maintains an array M that points to a two dimensional array of integers (zeros and ones). It is updated with each new character in the text window. A variable $error$ stores the number of mismatches and is updated for each character that is processed in the current window, depending on the value of vector M for the respective character.

The pseudocode of the algorithm is presented in Alg. 22. In the preprocessing phase, an array A is initialized to the number of occurrences of each character in the pattern. For each character c that occurs in the pattern, M_c (of size m) is assigned with A_c zero bits. The rest of the bits are assigned to ones specifying that $m - A_c$ is the count of characters different from c in P. For example, for the pattern $P = aabcd$, the value of M_a is 00111 as there are two occurrences of character a in the pattern.

During the search phase, the leftmost m characters are scanned from the text. The element ${}^*M_a$ represents the value of the base address of vector M_a. When a character a is read from the text window, ${}^*M_a$ is incremented by one, corresponding to the left shift operation of the binary vector. In the same way, when a character a is removed from the window, ${}^*M_a$ is decremented by one, corresponding to the shift right operation of the binary vector.

Let us consider an example having a pattern $P = agccacgc$ and a text

$T = tagcccgaatcgaccga$ over an alphabet $\Sigma = \{a, c, g, t\}$. In this example, we consider the value of $k = 0$. We count the occurrences of each character, obtaining $A_a = 2$, $A_c = 4$, $A_g = 2$, $A_t = 0$.

For each character of the alphabet c, M_c is a binary vector containing a sequence of $cc(P, c)$ zeros and $m - cc(P, c)$ ones. The values of the vectors M_a, M_c, M_g and M_t are represented as follows. Initially, the leftmost m characters of the text are scanned $[tagcccga]atcgaccga$. The first character to be processed is t at position t_0.

M_a	0	0	1	1	1	1	1	1
M_c	0	0	0	0	1	1	1	1
M_g	0	0	1	1	1	1	1	1
M_t	1	1	1	1	1	1	1	1

After reading the leftmost character, the variable $error$ (the number of mismatches) is updated to value 1, then the value of M_t is updated by left shifting as shown below.

M_t	1	1	1	1	1	1	1	0

The next character a at position t_1 is processed. The value of vector M_a is updated by left shifting it by one position as shown below. All characters in the first window are processed in the same manner.

M_a	0	1	1	1	1	1	1	0

After the processing of the current text window, the value of variable $error$ is one, as only the character t at the position t_0 is an overflow character. The condition ($error \leq k$) is not true for the first text window as $error = 1$ holds. Hence no occurrence of permuted pattern in the first text window is reported and the window is moved forward by one position. Every time the character c is processed, we update the value of vector M_c and value of variable $error$ and report an occurrence of the permuted pattern when $error = 0$. At the end, the total number of occurrences of the permuted pattern in the text is reported.

Algorithm 22 AJMS($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do A_c ← 0
 2: for j ← 0 to m − 1 do
 3:       A_{p_i} ← A_{p_i} + 1
 4: for all i ∈ Σ do
 5:       for j ← 0 to m − 1 do
 6:             M_{i,j} ← 1
 7:       if A_i > 0 then
 8:             for j ← 0 to A_i − 1 do
 9:                   M_{i,j} ← 0
    /* Searching */
10: error ← 0
11: for i ← 0 to m − 1 do
12:       error ← error + *M_{t_i}
13:       M_{t_i} ← M_{t_i} + 1
14: if error ≤ k then
15:       occ ← occ + 1
16: j ← 0
17: for i ← m to n − 1 do
18:       a ← t_j
19:       b ← t_i
20:       M_a ← M_a − 1
21:       error ← error + *M_b − *M_a
22:       M_b ← M_b + 1
23:       if error ≤ k then
24:             occ ← occ + 1
25:       j ← j + 1
26: return occ
```

Analysis

There are two main loops of the algorithm in search phase. The first loop reads the first m characters of the text and the second for loop reads the rest of the text. The complexity of this algorithm is $\Theta(n)$.

7.6 AJMRS

AJMRS (Approximate Jumbled Matching by Rotating Shifts) [35] is a forward type approximate algorithm for jumbled matching. It uses bitparallelism and circular rotation of the bitvector. It works by representing each character in the pattern as a bitvector of length m, which contains the number of zeros equal to the number of occurrences of each respective character. The zeros are inserted at the right end of the bitvector and the number of mismatches are calculated after each rotation. The text

window is searched for the occurrence of the permuted pattern and the window is moved forward by one step. AJMRS differs from AJMS in two ways. AJMS applies shifting and AJMRS applies circular shifting. The vectors are integer arrays in AJSM and computer words in AJMRS.

Because AJMS and AJRMS are forward type algorithms, we compare them with Count algorithm. In Count, a frequency counter C_c is maintained for every character c in the pattern, whereas in AJMS and AJRMS, for each character c of the pattern, a bitvector M_c is maintained containing a sequence of $cc(P, c)$ zeros and $m - cc(P, c)$ ones. The character counts are maintained (using the frequency counter in the Count algorithm and the frequency vectors in AJMS) for each character in the current text window. However, in AJRMS, the counters are updated for each text window by performing left and right shift rotations.

Algorithm 23 AJMRS($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Preprocessing */
 1: for all c ∈ Σ do A_c ← 0
 2: for j ← 0 to m − 1 do
 3:         A_{p_i} ← A_{p_i} + 1
 4: for all i ∈ Σ do
 5:         M_i ← 1^m << A_i
    /* Searching */
 6: error ← 0
 7: for i ← 0 to m − 1 do
 8:         error ← error + M_{t_i} & 1
 9:         RotateRight(M_{t_i})
10: if error ≤ k then
11:         occ ← occ + 1
12: j ← 0
13: for i ← m to n − 1 do
14:         a ← t_j
15:         b ← t_i
16:         RotateLeft(M_a)
17:         error ← error + (M_b & 1) − (M_a & 1)
18:         RotateRight(M_b)
19:         if error ≤ k then
20:                 occ ← occ + 1
21:         j ← j + 1
22: return occ
```

The pseudocode of the algorithm AJMRS is represented in Alg. 23. In the preprocessing phase, the frequency vector A is initialized to the frequency count of each character in the pattern. Array M is used as the state vector having a bitvector of ones followed by zeros (equal to the frequency of each character in the pattern).

In the searching phase, a counter $error$ is used to count the number of mismatches in the current text window. The first window of the text is processed first before processing the rest of the text. The value of variable $error$ is calculated with each character and the bit string is rotated towards the right. In the following steps, for the new character of the text window, the bit string is rotated left and the error count is updated. Thereafter, right rotation is performed for the new character. In the same manner, the whole text is scanned by moving the window forward by one step and reporting the total number of occurrences of the permuted pattern in the text.

Let us consider an example having a pattern $P = aaggtac$ and a text $T = cagtagatactga$ over an alphabet $\Sigma = \{a, c, g, t\}$ for $k = 0$. The algorithm reads the pattern and updates array M of each character of the pattern as $M_a = 1111000$, $M_c = 1111110$, $M_g = 1111100$ and $M_t = 1111110$ as represented by the following arrays, by left shifting the bit string by zeros (equal to the number of frequency of each character in the pattern). It processes the first window of the text ($[cagtaga]tactga$) before processing the rest of the text.

While processing the first window, the algorithm counts the number of errors at step 8 of the algorithm. Then, the algorithm rotates the vector M_c of the character in turn towards the right while reading the text window from left to right. There is no increment in the error count for character c. The bit sequence corresponding to the character c ($[\underline{c}agtaga]tactga$) is rotated towards the right, resulting in the bitvector represented as follows.

M_c	0	1	1	1	1	1	1

The error count is updated in the next step, then a right rotation operation is performed over the bit string for the next respective character. There is no addition in the error count for the next character a (at the position t_1) of the window as shown below. As the count in the current text window of character a is not more than the count of character a in the pattern.

M_a	0	1	1	1	1	0	0

The remaining characters of the text window are processed in the same manner. The number of mismatches is calculated first and then the right rotation is performed. The result of final rotation for the remaining characters of the first text window is represented as follows.

M_g	0	1	1	1	1	1	0
M_t	0	1	1	1	1	1	1
M_a	0	0	1	1	1	1	0
M_g	0	0	1	1	1	1	1
M_a	0	0	0	1	1	1	1

The number of mismatches for the current text window is zero. Hence an occurrence of the pattern is reported. In the following steps, the text window is moved forward by one position. The processing of the new text window begins by left rotation of the element M_c (for the first character c of the each window) and the right rotation of the element M_d (for the newly added character d). The total number of occurrences of the permuted pattern in the text is reported at the end.

Analysis

In the search phase, the algorithm works linearly by taking $\Theta(n)$ times to execute the main *for* loop in line 13.

7.7 AFB

Algorithm 24 AFB($P = p_0p_1 \cdots p_{m-1}, T = t_0t_1 \cdots t_{n-1}, k$)

```
    /* Preprocessing */
 1: X ← 0
 2: for i ← 0 to m − 1 do
 3:       X ← X + p_i − 48
    /* Searching */
 4: occ ← 0
 5: xa ← X − k; xb ← X + k
 6: for j ← 0 to m − 1 do
 7:       C ← C + t_j − 48
 8: while j < n do
 9:       if ((C ≥ xa) and (C ≤ xb)) then
10:             occ ← occ + 1
11:       C ← C − t_{j−m} + t_j
12:       j ← j + 1
13: return occ
```

AFB (Approximate Forward for Binary data) is an approximate version of EFB algorithm. It finds the approximate occurrences of the permuted pattern in binary data containing only ASCII characters 0 and 1. The main idea of this algorithm is to count the number of 1's and compare it with the number of 1's in each text window, taking in account the value

of k. If both the text window and the pattern contain the same number of 1's (considering number of k's), an occurrence of permuted pattern in the text is reported.

The pseudocode of AFB is presented in Alg. 24. The preprocessing of AFB is same as EFB. In the search phase, the only difference is the computation of constants xa and xb and the comparison in the main $while$ loop in line 8. Constants xa and xb express the acceptable minimum and maximum counts of 1's and are given values $X - k$ and $X + k$ respectively. The main $while$ loop compares variable C with xa and xb and updates variable C for each text window by moving forward one step each time. The and in line 9 can be implemented with the short circuit AND in the C language.

Let us consider an example having a pattern $P = 111$ and a text $T = 11001100$ over the alphabet $\Sigma = \{1, 0\}$ for $k = 1$. The value of variable X is 3, since the number of 1's in the pattern is 3. Hence, the values of xa and xb are given value 2 and 4 respectively. The first text window ($[110]01100$) is processed by counting the number of 1's and is stored in variable C. Thereafter C is compared with xa and xb. As the test is true, an occurrence is reported. The text window is moved forward by one position ($1[100]1100$) and the count of 1's is calculated which comes out as 1. Since the value of C is less than xa, the text window is moved forward without reporting a match and the value of C is updated to 2. In a similar manner, the whole text is processed and the total number of occurrences of the permuted pattern is reported in the end.

Analysis

The AFB algorithm is based on the Count algorithm. The main loop of AFB is in line 8. The algorithm scans the whole text by moving forward one position yielding the complexity of $\Theta(n)$.

8. Episode Matching with JPM

Mannila et al. [47] introduced the problem of episode matching. An episode is a collection of events that occur within a short time interval. This sequence of events needs to be analyzed in many areas of computer science such as data mining, bioinformatics and machine learning. Episode matching helps to analyze frequent events occurring together and get significant information from data. An event is a single incidence that occurs with a certain frequency. If the frequency of a group of closeby events is more than the threshold frequency, the group is considered as an episode. An important problem is to examine such events and to find the frequent episodes. The examples of such events can be, alarms in a telecommunication network, user interface actions, crimes committed by a person etc.

Episode matching can be defined as follows. Given a sequence S of certain events, a threshold frequency *fr* and a window width m, the task is to find all the maximal frequent episodes having frequency equal to *fr* or more in windows. In general, all events are associated with two parameters, event type (such as user actions) and time of occurrence of the event.

Parallel episode matching [47] is a type of episode matching in which there are no constraints on the relative order of distinct events. Parallel episode matching is closely related to the jumbled matching if we use characters to represent events. As in jumbled matching, we are interested in the occurrences of the characters in strings irrespective of the order of occurrence. In a similar way, episode matching focuses on the collection of the events that occur frequently in a close proximity. For example, two events X and Y occur in some sequence of events and their occurrence has significance if they occur together or close to each other and also form an episode with threshold frequency.

In our algorithm, we do not explicitly consider the time attribute

of the events. We assume that the events are sequential and each event (including a dummy event) takes a constant time. We only consider the occurrences of the events. For example, consider a text $T = bdeebcdcabaeeccebadadbaadbcfafdb$, with width $m = 5$ and threshold frequency $fr = 14$.

- For the episodes of length 1, there are four episodes that satisfy the conditions for episode matching. Here, a, b, c, d are episodes of length 1 and e, f are not episodes since their frequency is less than the threshold frequency.

- For the episodes of length 2, the candidates are ab, ac, ad, bc, bd and cd. Since e and f are not episodes, they cannot be a part of candidate episodes. The candidates ab, ad, bd and bc are episodes as they satisfy the above mentioned conditions. The candidates cd and ac are not episodes, as their frequency is less than fr.

- For the episodes of length 3, abd is a candidate episode and also an actual episode. abc and acd are not candidates, because ac is not an episode.

Hence abd and bc are maximal episodes and their subepisodes are also considered as episodes.

We introduce a new algorithm EPI for parallel episode matching. It finds the maximal frequent episodes from the given sequence. Alg. 26 represents pseudocode of the algorithm. In this algorithm, we process the text with a sliding window of m characters. We assume the characters representing events are from 0 to $w - 1$ where w is the word size. Each episode u is stored in the variable $Episode_u$ and the variable MAX contains the largest episode. A subroutine $WinConstr$ is invoked to create a descriptor for each window. Each descriptor is a bitvector, where each bit corresponds to a certain event. In $WinConstr$, the descriptors are formed using three successive loops, each for the first window, middle windows and the last window. The following operation adds an event x to the descriptor of a window.

$$win_j \leftarrow win_j \mid (1 << x)$$

In the descriptor, 1 represents the presence of that event in the window

and 0 represents that event is not present in the window. We count the number of windows where each event appears.

Algorithm 25 WinConstr$(t_0 t_1 \cdots t_{n-1}, m)$

```
1: for all c ∈ Σ do A_c ← 0; count_c ← 0; last_c ← −∞
2: win_n ← 0
3: for j ← 0 to m − 1 do
4:     x ← t_j; d ← j − last_x
5:     if d > j + 1 then
6:         d ← j + 1
7:     count_x ← count_x + d; last_x ← j; A_x ← A_x + 1
8:     if A_x = 1 then
9:         win_{m−1} ← win_{m−1} | (1 << x)
10: while j < (n − 1) − m do
11:     x ← t_{j−m}; win_j ← win_{j−1}; A_x ← A_x − 1
12:     if A_x = 0 then
13:         win_j ← win_j & ∼(1 << x)
14:     x ← t_j; d ← j − last_x
15:     if d > m then
16:         d ← m
17:     count_x ← count_x + d; last_x ← j
18:     if d ≥ m then
19:         d ← m
20:     else
21:         if (j − d + m) ≥ (n − 1) − 1 then
22:             d ← 0
23:     count_x ← count_x + d; A_x ← A_x + 1
24:     if A_x = 1 then
25:         win_j ← win_j & (1 << x)
26:     j ← j + 1
27: while j < n − 1 do
28:     x ← t_{j−m}; win_j ← win_{j−1}; A_x ← A_x − 1
29:     if A_x = 0 then
30:         win_j ← win_j & ∼(1 << x)
31:     x ← t_j; d ← j − last_x
32:     if d ≥ m then
33:         d ← n − 1 − j
34:     else
35:         if j − d + m ≥ n − 2 then
36:             d ← 0
37:         else
38:             d ← (n − 1) − j − m + d
39:     count_x ← count_x + d; last_x ← j; A_x ← A_x + 1
40:     if A_x = 1 then
41:         win_j ← win_j | (1 << x)
42:     j ← j + 1
```

Algorithm 26 EPI($T = t_0 t_1 \cdots t_{n-1}, m, \mathit{fr}$)

```
1: z ← 0
2: WinConstr(t_0 t_1 ··· t_{n-1}, m)
3: Checkepisode(0, 0, fr)
4: for u ← 0 to z − 1 do
5:     MAX ← Episode_u
6:     if MAX > 0 then
7:         for v ← 0 to z − 1 do
8:             if MAX < Episode_v then
9:                 MAX ← Episode_v
10:        if MAX ≠ 0 then
11:            Report Episode MAX
12:        for v ← 0 to z − 1 do
13:            if (MAX & Episode_v) = Episode_v then
14:                Episode_v ← 0
```

Algorithm 27 Checkepisode($Mask, index, \mathit{fr}$)

```
1: counter ← 0
2: for i ← index to Z − 1 do
3:     if count_i ≥ fr then
4:         New ← Mask | (1 << i)
5:         counter ← 0
6:         for j ← m − 1 to n − 1 do
7:             if (win_j & New) = New then
8:                 counter ← counter + 1
9:         if counter ≥ fr then
10:            Episode_z ← New
11:            z ← z + 1
12:        Checkepisode(New, i + 1, fr)
```

After performing the above operation for all the $n - m + 1$ windows, a subroutine $Checkepisode$ is invoked to find all the episodes. We consider the number of actual distinct events to be Z. The counts of each event are compared against the threshold frequency to find out whether their value is more than the threshold frequency. In line 10, we store each candidate episode, where the count of all the events present in the candidate is greater than or equal to the threshold frequency. We go through all the bits in a descriptor, starting from the lowest bit. If the frequency of the corresponding event is greater than or equal to the threshold frequency, then variable New is updated by setting the bit as 1 for that event, and $Checkepisode$ invokes itself to determine, if the event at the next position is an episode or not. To determine whether the candidate episode is an actual episode, the following operation is performed. It compares the presence of the events in the candidate episode with the events present in each window of the text.

$$if(win_j \ \& \ New) = New$$

If the comparison returns true, then the variable $counter$ is incremented by one to determine the total count of all such true comparisons. The value of the variable $counter$ is compared against the threshold frequency (***fr*** in $Checkepisode$). If the value of the $counter$ is greater than the threshold frequency, it means that the candidate episode is an actual episode. All the actual episodes are stored in an array of descriptors called $Episode$. It is to be noted that these episodes are not necessarily maximal episodes.

Thereafter, we find the maximal episodes among all the episodes. Hence forth, we look for the largest episode and its sub episodes in an array of descriptors $Episode$. After finding the largest episode (in terms of alphabetical order), we report it. Then its descriptor and the descriptors of its subepisodes are set to zero. In the same manner the whole array $Episode$ is scanned and all the maximal episodes are reported.

As in the previous example, let us consider text $T = bdeebcdcabaeeccebfadfadbaadbcfafdb$ of length 34 over six types of events a, b, c, d, e, f. We consider a window of width 5. There are 30 windows of width 5. All windows are processed by the subroutine $WinConstr$. For the first window $bdeeb$, the descriptor is 010110 which denotes that the events a, c, f are not present in the window and the events b, d, e are present in the window. The least significant bit corresponds to event a, the second least significant bit corresponds to event b and so on.

In a similar way, all other windows are represented as descriptors. Element A_c counts the number of event c in the current window. The array $count$ shows the count of windows where an event appears and the array $last$ represents the previous occurrence of an event. After constructing the descriptors of the windows, subroutine $Checkepisode$ is called. In $Checkepisode$, all the episodes i.e a, ab, abd, ad and so on, are stored in the array $Episode$. After finding all the episodes, the execution returns to the main EPI algorithm. In EPI, all the maximal episodes are computed and reported by eliminating all the subepisodes of maximal episodes. So the maximal episodes that are reported in our example are abd and bc.

Analysis

Let the number of actual distinct events be Z. Though the upper limit for the number of episodes reduces with the window size, we consider the rough upper limit as 2^Z. In the best case, the time complexity is $O(n+Z)$. Subroutine $WinConstr$ in EPI takes n times to create the descriptors for each window of length m. EPI then calls subroutine $Checkepisode$, which takes $O(Z)$ time in the best case, as the if condition in line 3 of $Checkepisode$ is always false (in the best case).

In the worst case (considering threshold frequency as 1), the complexity is $O(n + n2^Z + 2^{2Z})$. The creation of descriptors of all windows requires n steps. But in the worst case, there can be 2^Z candidate episodes (since in a set of x elements, there can be 2^x subsets). Hence, the subroutine $Checkepisode$ takes $O(n2^Z)$ times to find the actual episodes. Since in the worst case, the number of actual episodes is 2^Z, then each for loop in line 3 and 6 of EPI takes 2^Z. It results in $O(2^{2Z})$ time. Hence the complexity is $O(n + n2^Z + 2^{2Z})$.

9. Experiments

In this chapter, we present the experimental results of the algorithms. The tests were run on an Intel 2.70 GHz i7 processor with 16 GB of memory. All the algorithms were implemented in the C programming language and run in 64-bit mode in the testing framework of Hume and Sunday [36]. We used four types of data (English, protein, DNA, binary) for testing the algorithms. The protein text is 3.3 MB long, English text (KJV Bible) is 4.0 MB long, binary text is 4.0 MB long and DNA text is 4.5 MB long. All the tests were taken from the Smart corpus [25] except binary data. We have used randomly generated binary text.

For each text, we had six sets of 200 patterns with lengths: $m = 5, 10, 20, 30, 50$ and 100 picked randomly from the text. The results were obtained as an average of nine runs. The best time for each combination of m and k has been boxed. The test results with SIMD algorithms are shown in the later part of the chapter. The texts were run with 200 patterns of lengths $m = 4, 5, \cdots, 10$ picked from each text. All the line graphs in this chapter show the comparison of execution times of the best five algorithms corresponding to the results of immediate previous tables. We have not shown the results of GFG [29] because they were mostly slower than BAM in tests of [11].

9.1 Exact Jumbled Matching Without SIMD

Table 9.1 shows the results for English data for exact jumbled matching. From the table, it can be seen that BAM2 is the fastest for all pattern lengths. For $m = 4, 5, \cdots, 9$ we get better results with SIMD algorithms that are shown in Tables 9.10 and 9.12.

Table 9.1. Execution times of algorithms (in seconds) for exact JPM for English data.

m	PBA	SBA	ASM	AIDM	DC	Count	Mcount	BAM
5	6.357	3.195	4.833	5.652	5.454	2.471	2.214	1.819
10	7.323	3.145	5.998	6.543	6.621	2.472	2.211	0.861
20	7.245	3.196	6.284	7.232	6.634	2.481	2.213	0.575
30	6.997	3.196	6.346	7.343	6.640	2.477	2.213	0.455
50	6.482	3.194	6.551	7.457	5.625	2.477	2.214	—
100	5.762	3.198	6.652	6.652	4.647	2.479	2.212	—

m	AJSM	AJMRS	BAMs	BAM2	ABAM	EBL	AF	ABL
5	3.933	5.429	1.275	0.729	1.388	0.734	1.815	1.864
10	3.935	5.431	0.873	0.314	1.028	0.633	1.815	1.071
20	3.934	5.432	0.582	0.248	0.684	0.543	1.815	0.705
30	3.933	5.434	0.439	0.253	0.512	0.482	1.815	0.532
50	3.932	5.432	0.299	0.237	—	0.519	1.815	0.413
100	3.931	5.433	0.199	0.155	–	0.356	1.815	0.342

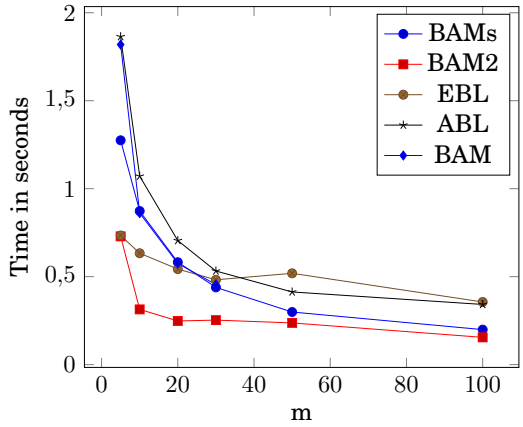


Figure 9.1. Execution times of algorithms (in seconds) for exact JPM for English data.

In Fig. 9.1, the comparison of algorithms for exact matching for English data is shown. It is clearly visible that BAM2 is fastest for all values of m.

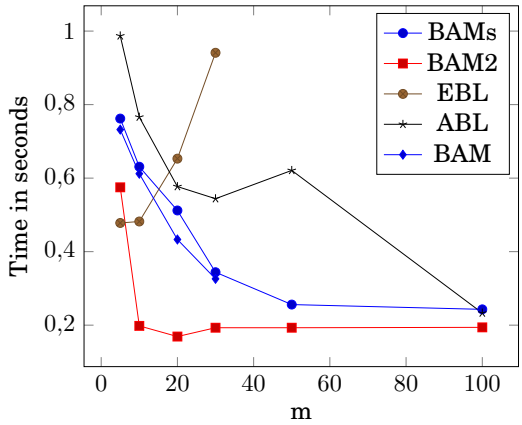


Table 9.2. Execution times of algorithms (in seconds) for exact JPM for protein data.

m	PBA	SBA	ASM	AIDM	DC	Count	Mcount	BAM
5	4.791	2.213	3.751	3.771	3.423	1.933	1.773	0.732
10	5.147	2.214	4.225	4.613	4.435	1.932	1.772	0.612
20	4.781	2.215	4.454	5.026	5.067	1.933	1.773	0.433
30	4.377	2.211	4.508	5.131	4.924	1.932	1.772	0.326
50	3.892	2.222	4.561	5.315	4.417	1.933	1.772	—
100	3.355	2.212	4.582	5.373	3.361	1.933	1.772	—

m	AJSM	AJMRS	BAMs	BAM2	ABAM	EBL	AF	ABL
5	2.817	3.903	0.762	0. 575	0.862	0.478	1.512	0.987
10	2.816	3.905	0.512	0.169	0.522	0.653	1.512	0.577
30	2.818	3.895	0.344	0.193	0.424	0.941	1.512	0.544
50	2.819	3.907	0.256	0.193	—	—	1.515	0.631
100	2.815	3.903	0.243	0.194	—	—	1.514	0.233

Figure 9.2. Execution times of algorithms (in seconds) for exact JPM for protein data.

Table 9.2 presents the results for protein data for exact jumbled matching. From the table, it is evident that EBL is the fastest for $m = 5$ and as m becomes greater than 5, BAM2 outperforms EBL. The results in Table 9.2 for protein data do not differ much from Table 9.1, but EBL is slower on proteins due to the uniformity of the data. EBL is very slow for $m > 30$ (results not shown), because then the number of forbidden characters is low. Fig. 9.2 has similar results as Fig. 9.1, but for $m = 5$, EBL is the fastest. For $m = 4, 5$ and 6 we get better results with SIMD algorithms and are shown in Tables 9.11 and 9.13.

Table 9.3. Execution times of algorithms (in seconds) for exact JPM for DNA data.

m	PBA	SBA	ASM	AIDM	DC	Count	Mcount	BAM
5	7.861	3.642	7.861	8.491	9.749	2.724	2.413	3.287
10	7.121	3.642	7.844	8.712	9.312	2.721	2.411	2.891
20	6.452	7.771	7.564	8.661	9.122	2.723	2.409	2.475
30	6.091	7.723	7.449	8.612	8.931	2.727	2.411	2.099
50	5.698	3.698	7.682	8.581	8.705	2.724	2.411	2.071
100	5.339	3.642	7.786	8.534	8.474	2.724	2.411	2.276

m	AJMS	AJMRS	BAMs	BAM2	ABAM	EFS	ABS	AF
5	4.513	6.291	3.283	1.511	3.981	1.153	3.961	2.194
10	5.511	6.313	2.685	1.768	3.468	1.158	3.432	2.194
20	4.496	6.314	2.268	1.657	3.177	1.154	3.236	2.194
30	4.483	6.345	1.974	1.458	2.898	1.161	2.911	2.193
50	4.499	6.312	1.972	1.343	3.068	1.151	3.002	2.194
100	4.486	6.313	2.212	1.268	3.641	1.142	3.411	2.194

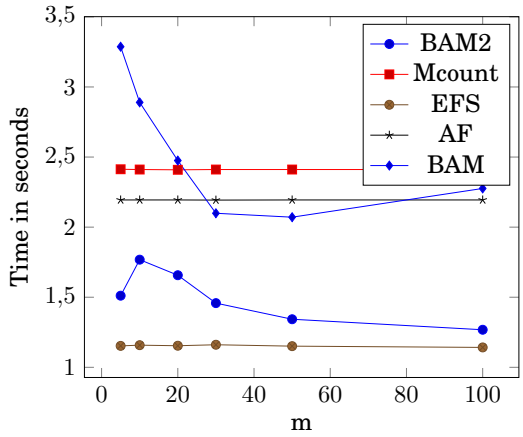


Figure 9.3. Execution times of algorithms (in seconds) for exact JPM for DNA data.

Table 9.3 depicts the results for DNA data for exact jumbled matching. From the table it can be noticed that EFS is the fastest for all pattern lengths and is at least two times faster when compared with the previous algorithms. Observe also that BAM2 is faster than BAM with a wide margin. We have not included the test results of AIDM for approximate JPM, as it was not competitive. Fig. 9.3 depicts that EFS is a clear winner for all vales of m.

Table 9.4. Execution times of algorithms (in seconds) for exact JPM for binary data.

m	PBA	SBA	ASM	DC	Count	Mcount	BAM
5	6.365	3.273	6.423	6.651	2.348	2.340	5.380
10	6.515	3.272	6.514	7.883	2.356	2.345	6.438
20	6.464	3.268	6.466	8.253	2.351	2.345	7.765
30	4.441	3.273	6.443	8.325	2.348	2.346	8.877
50	6.422	3.273	6.423	8.397	2.350	2.344	10.843
100	6.410	3.271	6.405	8.473	2.347	2.344	15.546

m	AJMS	AJMRS	BAMs	BAM2	ABAM	EFS	ABS	AF	EFB
5	3.916	5.553	5.523	2.687	7.631	0.964	7.241	2.092	0.914
10	3.894	5.561	6.281	4.270	9.212	0.958	8.798	2.070	0.928
20	3.894	5.552	8.076	5.671	11.541	0.963	11.683	2.066	0.928
30	3.890	5.554	9.358	6.736	14.321	0.987	14.321	2.066	0.928
50	3.895	5.553	11.654	7.742	18.228	0.991	17.762	2.066	0.928
100	3.910	5.554	15.591	9.665	26.112	0.960	24.511	2.068	0.930

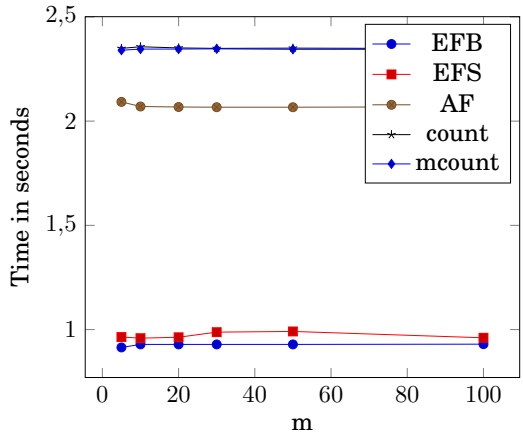


Figure 9.4. Execution times of algorithms (in seconds) for exact JPM for binary data.

Table 9.4 shows the results for binary data for exact jumbled matching. It is clear from the table that EFB is the fastest for all pattern lengths. Fig. 9.4 shows the corresponding results. It is to be noted that we adapted the EFS algorithm to the binary alphabet.

9.2 Approximate Jumbled Matching

Table 9.5. Execution times of algorithms (in seconds) for approximate JPM for English data.

m	k	ASM	DC	Count	Mcount	AJMS	AJMRS	ABAM	AF	ABL
5	1	5.571	5.382	2.372	2.175	3.934	5.434	3.451	1.813	4.056
10	1	5.918	6.984	2.371	2.175	3.928	5.434	1.843	1.814	2.214
20	1	6.282	7.821	2.373	2.175	3.928	5.432	1.051	1.814	1.257
30	1	6.373	7.633	2.372	2.175	3.937	5.428	—	1.815	1.001
50	1	6.432	6.347	2.371	2.175	3.934	5.431	—	1.815	0.776
100	1	6.412	5.119	2.372	2.175	3.922	5.432	—	1.815	0.732
5	2	5.481	5.582	2.371	2.175	3.933	5.433	6.657	1.815	8.952
10	2	6.102	7.493	2.372	2.175	3.927	5.432	3.328	1.814	5.025
20	2	6.295	8.277	2.372	2.175	3.926	5.432	1.553	1.8144	2.557
30	2	6.351	7.834	2.372	2.175	3.927	5.432	—	1.815	1.979
50	2	6.449	6.767	2.372	2.175	3.927	5.425	—	1.814	1.618
100	2	6.443	5.662	2.372	2.175	3.925	5.434	—	1.814	1.613
5	3	5.483	4.791	2.372	2.175	3.925	5.435	8.752	1.814	14.276
10	3	5.993	7.543	2.372	2.175	3.923	5.433	5.902	1.814	10.667
20	3	6.289	8.461	2.372	2.175	3.924	5.431	—	1.814	5.788
30	3	6.342	8.168	2.371	2.175	3.925	5.432	—	1.814	4.535
50	3	6.449	7.152	2.372	2.175	3.926	5.425	—	1.814	3.601
100	3	6.442	5.876	2.372	2.175	3.925	5.433	—	1.815	3.535

Table 9.5 shows the results for English data for approximate jumbled matching. For $k = 1, 2$ and 3, AF is the fastest for $m = 5, 10$. As an exception, ABAM is the fastest for $k = 1$ and $m = 20$. Thereafter, for $m = 30, 50$ and 100, ABL comes out to be the fastest. For $k = 2$, the result is same

as for $k = 1$ except at $m = 30$, where AF performs the best. However, for $k = 3$, AF is the fastest for all values of m.

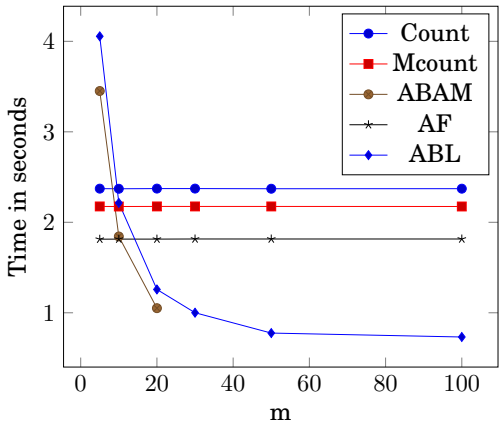


Figure 9.5. Execution times of algorithms (in seconds) for approximate JPM for English data for $k = 1$.

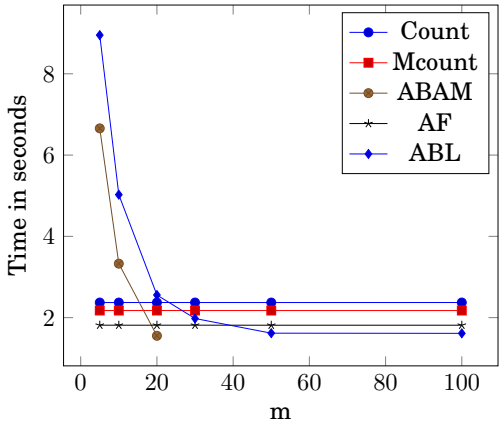


Figure 9.6. Execution times of algorithms (in seconds) for approximate JPM for English data for $k = 2$.

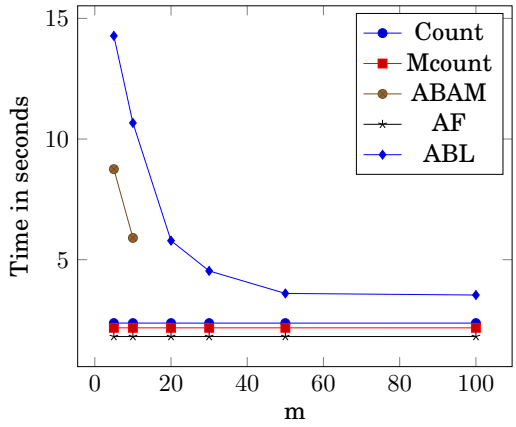


Figure 9.7. Execution times of algorithms (in seconds) for approximate JPM for English data for $k = 3$.

Fig. 9.5, 9.6 and 9.7 show the result of execution times of algorithms for approximate matching for English data for $k = 1, 2$ and 3 respectively. The results for $k = 1$ and 2 are similar. For smaller values of m, AF is the fastest. However, for $m = 20$, ABAM performs the best. For $k = 2$, and larger patterns, ABL is the fastest except for $m = 30$, where AF has the least execution time. For $k = 3$, AF is a clear winner for all values of m.

Table 9.6. Execution times of algorithms (in seconds) for approximate JPM for DNA data.

m	k	ASM	DC	Count	Mcount	AJMS	AJMRS	ABAM	AF	ABS
5	1	7.848	8.091	2.711	2.564	4.522	6.321	7.778	2.215	7.567
10	1	7.841	9.513	2.713	2.564	4.513	6.322	7.156	2.215	7.244
20	1	7.766	9.586	2.713	2.563	4.499	6.319	6.356	2.215	6.112
30	1	7.712	9.451	2.712	2.563	4.531	6.311	5.221	2.215	5.099
50	1	7.675	9.166	2.713	2.563	4.461	6.305	4.857	2.216	4.671
100	1	7.641	8.801	2.712	2.564	4.456	6.301	4.695	2.216	4.517
5	2	7.861	8.267	2.714	2.563	4.503	6.316	8.424	2.216	7.954
10	2	7.823	9.405	2.712	2.564	4.515	6.311	10 .235	2.215	8.187
20	2	7.711	9.536	2.713	2.563	4.493	6.308	11.871	2.215	7.778
30	2	7.678	9.458	2.713	2.563	4.510	6.305	8.564	2.215	7.325
50	2	7.662	9.393	2.713	2.563	4.494	6.303	7.118	2.215	7.157
100	2	7.638	9.081	2.713	2.564	4.485	—	6.887	2.215	6.564
5	3	8.586	7.767	2.712	2.564	4.501	6.315	8.444	2.216	8.345
10	3	7.863	9.114	2.724	2.564	4.498	6.317	14 .381	2.214	9.486
20	3	7.845	8.878	2.713	2.564	4.502	6.314	16.117	2.214	9.117
30	3	7.753	8.752	2.713	2.563	4.479	6.307	13.567	2.215	8.661
50	3	7.677	8.557	2.714	2.563	4.457	6.305	11.812	2.215	8.592
100	3	7.648	8.224	2.714	2.563	4.452	—	9.557	2.214	8.365

From the results of DNA data in Table 9.6 it can be clearly seen that AF is the fastest for all values of m and k.

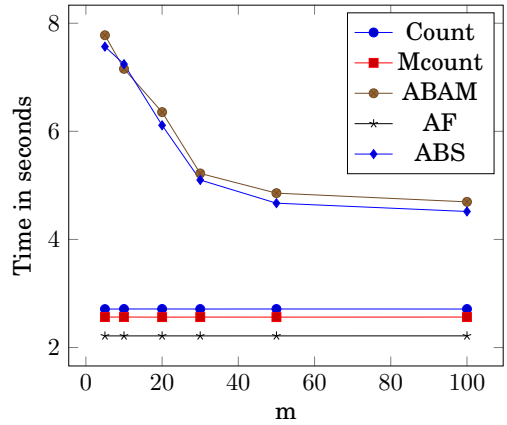


Figure 9.8. Execution times of algorithms (in seconds) for approximate JPM for DNA data for $k = 1$.

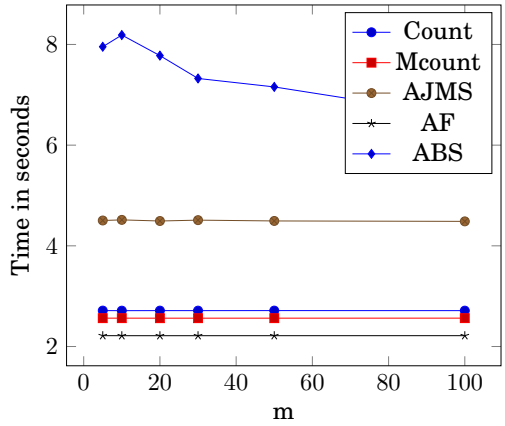


Figure 9.9. Execution times of algorithms (in seconds) for approximate JPM for DNA data for $k = 2$.

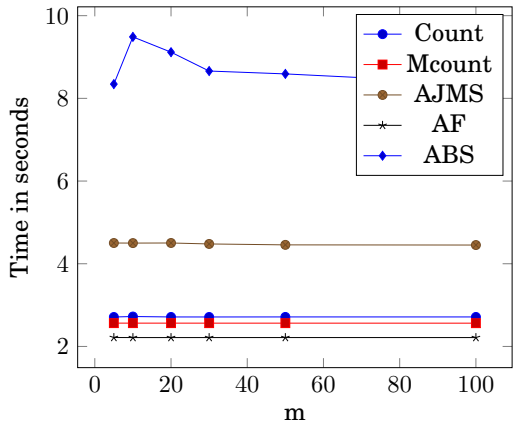


Figure 9.10. Execution times of algorithms (in seconds) for approximate JPM for DNA data for $k = 3$.

Fig. 9.8, 9.9 and 9.10 show the result of execution times of algorithms for approximate matching for DNA data for $k = 1, 2$ and 3 respectively. It is clearly evident from the graphs that AF is the clear winner for all values of k and m.

In Table 9.7, the results for protein data for approximate jumbled matching are shown. For $k = 1$, AF is the fastest for $m = 5$, but as m approaches 10, ABAM is the fastest. However, as m reaches 50, AF outperforms all other algorithms. For $k = 2$, AF is the fastest except for $m = 20$ where ABAM is the fastest. As the value of k reaches 3, AF is the fastest for all values of m.

Table 9.7. Execution times of algorithms (in seconds) for approximate JPM for protein data.

m	k	ASM	DC	Count	Mcount	AJMS	AJMRS	ABAM	AF	ABL
5	1	3.762	3.559	1.932	1.762	3.211	3.923	2.131	1.512	2.352
10	1	4.237	4.814	1.933	1.762	3.211	3.924	1.242	1.515	1.477
20	1	4.456	5.802	1.934	1.762	3.212	3.923	0.791	1.513	1.075
30	1	4.508	5.674	1.932	1.762	3.211	3.923	0.630	1.53	1.119
50	1	4.565	5.864	1.932	1.762	3.212	3.923	—	1.535	1.576
100	1	4.583	4.168	1.931	1.763	3.211	3.923	—	1.514	3.487
5	2	3.755	3.629	1.932	1.761	3.211	3.923	4.302	1.514	5.168
10	2	4.223	5.077	1.924	1.762	3.211	3.923	2.019	1.514	2.875
20	2	4.466	6.127	1.934	1.767	3.211	3.922	1.101	1.513	2.075
30	2	4.548	6.029	1.934	1.765	3.212	3.923	—	1.513	2.299
50	2	4.584	5.446	1.932	1.763	3.210	3.923	—	1.513	3.388
100	2	4.986	4.548	1.932	1.762	3.210	3.923	—	1.511	3.166
5	3	3.767	3.464	1.933	1.762	3.211	3.923	6.519	1.512	8.766
10	3	4.225	5.135	1.939	1.762	3.211	3.923	3.331	1.512	5.915
20	3	4.456	6.185	1.932	1.763	3.211	3.924	—	1.535	4.338
30	3	4.503	6.191	1.931	1.763	3.211	3.923	—	1.531	4.755
50	3	4.563	5.634	1.933	1.763	3.211	3.923	—	1.515	6.766
100	3	4.579	4.848	1.933	1.763	3.211	3.923	—	1.515	7.272

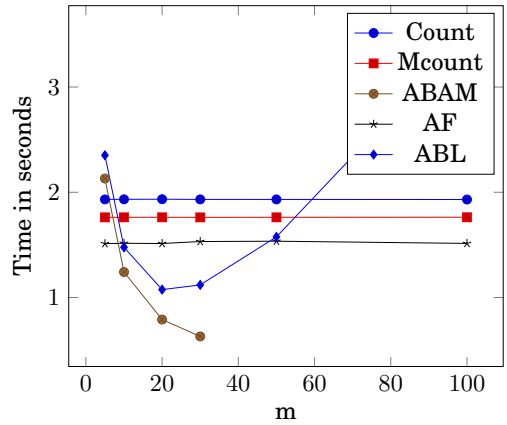


Figure 9.11. Execution times of algorithms (in seconds) for approximate JPM for protein data for $k = 1$.

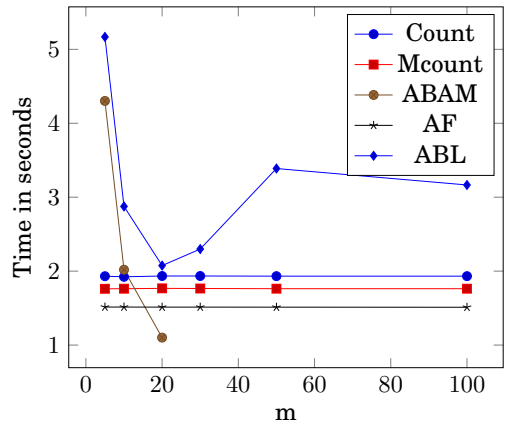


Figure 9.12. Execution times of algorithms (in seconds) for approximate JPM for protein data for $k = 2$.

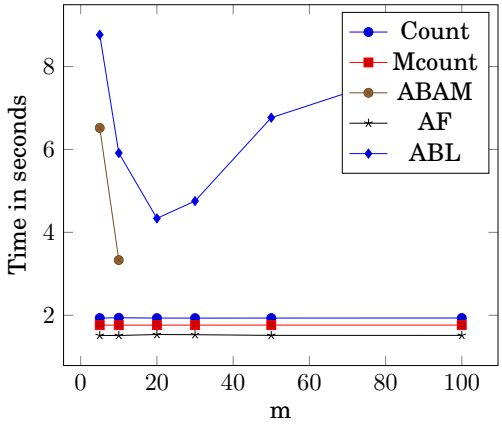


Figure 9.13. Execution times of algorithms (in seconds) for approximate JPM for protein data for $k = 3$.

Fig. 9.11, 9.12 and 9.13 show the result of execution times of algorithms for approximate jumbled matching for protein data for $k = 1, 2$ and 3 respectively. The results for protein data are similar to English data for $k = 1, 2$ and 3 except for larger patterns where AF is the fastest.

Table 9.8. Execution times of algorithms (in seconds) for approximate JPM for binary data.

m	k	ASM	DC	Count	Mcount	AJMS	AJMRS	ABAM	AF	ABS	AFB
5	1	6.341	4.525	2.346	2.349	3.907	5.557	8.160	2.041	7.531	1.494
10	1	6.517	6.087	2.348	2.347	3.898	5.553	11.695	2.032	11.436	1.411
20	1	6.460	6.270	2.347	2.345	3.890	5.553	16.390	2.028	15.213	1.336
30	1	6.445	6.452	2.346	2.345	3.873	5.567	20.641	2.030	19.412	1.222
50	1	6.438	6.446	2.348	2.345	3.872	5.552	27.653	2.028	25.661	1.155
100	1	6.412	6.951	2.346	2. 346	3.874	5.557	38.652	2.028	35.563	1.071
5	2	6.359	4.491	2.358	2.346	3.892	5.555	7.693	2.031	7.135	1.257
10	2	6.540	6.091	2.345	2.347	3.887	5.558	12.714	2.031	12.543	1.177
20	2	6.521	6.389	2.350	2.340	3.890	5.558	19.680	2.032	19.032	1.236
30	2	6.426	6.711	2.350	2.344	3.890	5.554	25.737	2.024	24.231	1.196
50	2	6.413	6.916	2.346	2.346	3.892	5.628	34.767	2.031	32.512	1.146
100	2	6.332	7.111	2.348	2.345	3.892	5.628	50.774	2.030	46.235	1.045
5	3	6.342	3.166	2.347	2.348	3.890	5.558	6.981	2.011	6.924	1.185
10	3	6.451	3.460	2.357	2.344	3.893	5.616	12.578	2.031	12.435	1.124
20	3	6.432	4.450	2.341	2.341	3.882	5.554	22.221	2.030	20.012	1.195
30	3	6.410	5.343	2.345	2.348	3.888	5.612	30.764	2.034	27.784	1.191
50	3	6.407	5.940	2.347	2.344	3.892	5.553	44.840	2.028	38.047	1.152
100	3	6.401	6.344	2.345	2.346	3.888	5.554	61.826	2.030	56.553	1.063

Table 9.8 presents the results for binary data for approximate jumbled matching. It is clear from the results that for all values of k and m, AFB outperforms all algorithms. Fig 9.14, 9.15 and 9.16 show the results corresponding to Table 9.8 for best five algorithms for $k = 1, 2$ and 3 respectively.

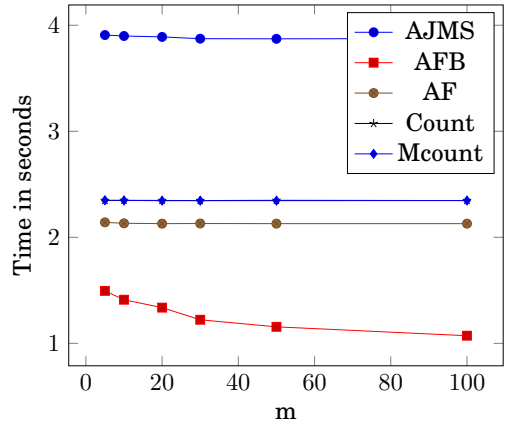


Figure 9.14. Execution times of algorithms (in seconds) for approximate matching for binary data for $k = 1$.

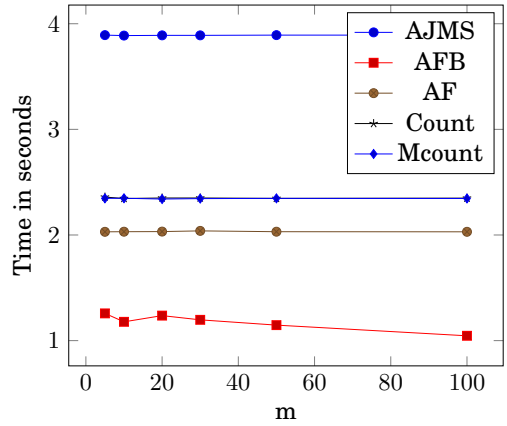


Figure 9.15. Execution times of algorithms (in seconds) for approximate matching for binary data for $k = 2$.

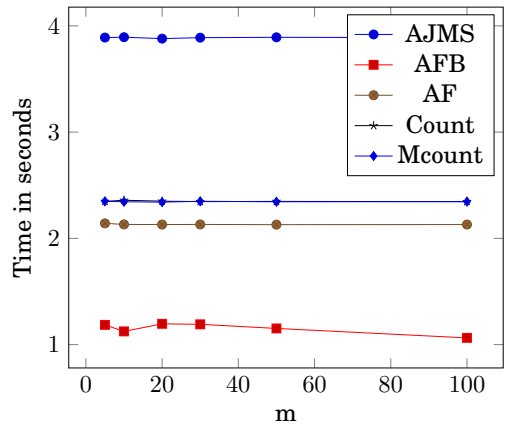


Figure 9.16. Execution times of algorithms (in seconds) for approximate matching for binary data for $k = 3$.

9.3 Exact Jumbled Matching With SIMD

The architecture of the processor determines the performance of SIMD instructions. The performance of a single instruction is measured by latency and throughput. Latency is the number of cycles taken by the processor to give the desired outcome from the given input. Throughput refers to the number of cycles between subsequent calls of the same instruction. We used processors Intel i7-860 and i5-4250U in our experiments. Their microarchitectures are Nehalem and Haswell [38], respectively. The latency and throughput of the SIMD instructions used in our algorithms for these processors are given in Table 9.9. One should observe that string matching instructions are slower than ordinary SIMD instructions. For other processors the difference may be still larger. Therefore which SIMD instructions an algorithm designer selects for his code is crucial.

Table 9.9. Latency and throughput of SIMD instructions for Nehalem and Haswelll [39].

Architecture	SIMD instruction	Latency	Throughput
Nehalem	$\_mm\_cmpistrm$	8	2
	$\_mm\_extract\_epi16$	3	1
	$\_mm\_cmpeq\_epi8$	1	0.5
	$\_mm\_movemask\_epi8$	1	1
Haswell	$\_mm\_cmpistrm$	11	3
	$\_mm\_extract\_epi16$	3	1
	$\_mm\_cmpeq\_epi8$	1	0.5
	$\_mm\_movemask\_epi8$	3	1

Tables 9.10 and 9.11 present the average execution times (for exact algorithms) for EA and LF algorithm schemes in seconds for English and protein data, respectively on Nehalem (which is our main test machine). We tested the EA and LF algorithm schemes with BAM, BAM2 and EBL as the checking subroutine, i.e. six new algorithms. From these, we selected LF-BAM2, EA-BAM2 and EA-EBL for further consideration. BAM, BAM2 and EBL were our reference methods. The running time of the PP algorithm for the EA scheme was about 10 ms.

In our tests, we applied BAM2 without the bin sharing technique. For the patterns having at most 10 characters we are not required to use shared bins in the 64-bit architecture. We used five bits for each bin.

In Table 9.10, the execution time of the best one of the new algorithms is 17–33 percent less than the best time for earlier algorithms for $4 \leq m \leq 9$. EA-BAM2 is the fastest for the short patterns of length 4 and 5. LF-BAM2 performs the best for the remaining pattern lengths except 10,

where BAM2 has the best execution time.

Table 9.10. Execution times of algorithms (in seconds) for exact JPM for English data on Nehalem.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	1.1918	1.4221	0.7274	0.6119	0.4995	0.5344
5	1.1942	0.7561	0.7364	0.5732	0.4903	0.5603
6	1.1037	0.5473	0.6891	0.4515	0.4859	0.5321
7	1.0116	0.4207	0.6502	0.3611	0.4809	0.5114
8	0.9521	0.3711	0.6267	0.3431	0.4761	0.5073
9	0.9035	0.3217	0.6257	0.2686	0.4791	0.5031
10	0.8671	0.3005	0.6345	0.3133	0.4803	0.4945

Table 9.11. Execution times of algorithms (in seconds) for exact JPM for protein data on Nehalem.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	0.6772	0. 9947	0.4573	0.5689	0.3307	0.3431
5	0.6913	0.5511	0.4245	0.5262	0.3203	0.3119
6	0.6567	0.4255	0. 3915	0.4629	0.3214	0.3221
7	0.5942	0.3167	0.3718	0.4065	0.3203	0.3341
8	0.5732	0.2545	0.3511	0.4115	0.3315	0.3472
9	0.5512	0.2025	0.3614	0.3405	0.3411	0.3512
10	0.5441	0.1798	0.4013	0.3792	0.3497	0.3623

The results in Table 9.11 for protein data show that the EA algorithm scheme is competitive in comparison with the previous algorithms. The EA scheme works best in cases of short patterns of length less than 7. EA-BAM2 is the fastest for pattern lengths 4 and 6. EA-EBL performs the best for $m = 5$. For the remaining pattern lengths, BAM2 performs the best. The LF algorithm scheme was not competitive for protein data.

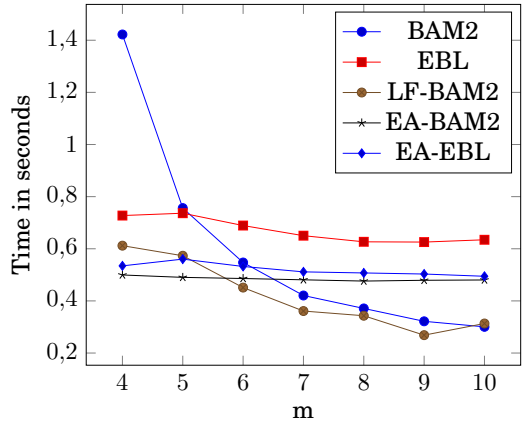


Figure 9.17. Execution times of algorithms (in seconds) for exact JPM for English data on Nehalem.

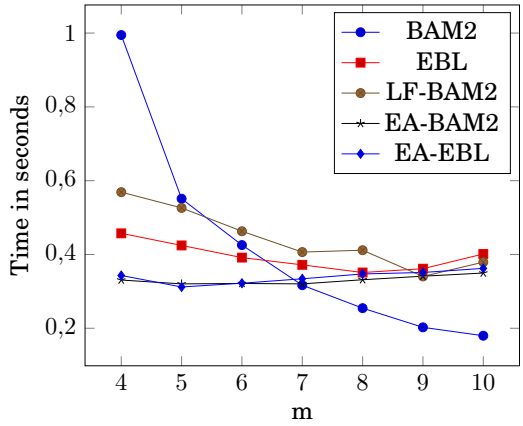


Figure 9.18. Execution times of algorithms (in seconds) for exact JPM for protein data on Nehalem.

Fig. 9.17 and 9.18 show the result of execution times for exact algorithms for English and protein data respectively on Nehalem. It is clear from the Fig. 9.17 that LF-BAM2 is the fastest for $m = 6, 7, 8, 9$. But for patterns smaller than $m = 6$, EA-BAM2 works the best. For the pattern $m = 10$, BAM2 has least execution time. For protein data, EA-BAM2 and EA-EBL are competitive and fastest for shorter patterns. For the patterns larger than $m = 6$, BAM2 is the clear winner.

Table 9.12. Execution times of algorithms (in seconds) for exact JPM for English data on Haswell.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	1.8640	2.3047	1.0800	0.7608	0.8158	0.8468
5	1.8643	1.1992	0.9686	0.7261	0.8082	0.9297
6	1.7281	0.7698	0.9083	0.5581	0.8082	0.9297
7	1.5944	0.6541	0.8801	0.4401	0.7945	0.9921
8	1.3017	0.3906	0.7258	0.3921	0.7192	0.8114
9	1.2395	0.3886	0.7490	0.2929	0.7294	0.8555
10	1.1960	0.3356	0.7636	0.3552	0.7247	0.8994

We also performed the same tests on a 1.3 GHz Haswell processor (i5-4250U) with 8 GB of memory, for English and protein data. The results are shown in Tables 9.12 and 9.13. In Table 9.12, the LF-BAM2 algorithm is a clear winner for all the pattern lengths except for $m = 8$ and 10. For certain pattern lengths, such as $m = 4, 5, 6$, the speedup is more than twenty percent. For protein data in Table 9.13, EBL and BAM2 are the winners. However, LF-BAM2 is better than BAM2 for $m = 4, 5$ and better than EBL for $m = 6, \cdots, 10$.

The Haswell processor has AVX2, which enables 32-byte SIMD computation. We compared the 32-byte version of LF-BAM2 with the 16-byte

version for pattern lengths $m = 4, 5, \cdots, 16$. In every case, the 16-byte version was slightly faster.

Table 9.13. Execution times of algorithms (in seconds) for exact JPM for protein data on Haswell.

m	BAM	BAM2	EBL	LF-BAM2	EA-BAM2	EA-EBL
4	0.8944	1.0338	0.4533	0.4776	0.4978	0.4912
5	0.8112	0.6251	0.4096	0.4519	0.4872	0.4880
6	0.7776	0.3541	0.4067	0.3870	0.4929	0.5227
7	0.7430	0.3230	0.4076	0.3467	0.5000	0.5583
8	0.7228	0.2255	0.4078	0.3549	0.5036	0.5676
9	0.7066	0.2271	0.4234	0.3103	0.5096	0.5889
10	0.6952	0.1809	0.4540	0.3478	0.5153	0.6388

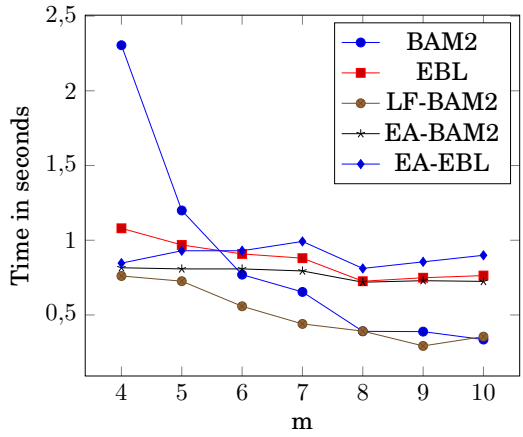


Figure 9.19. Execution times of algorithms (in seconds) for exact JPM for English data on Haswell.

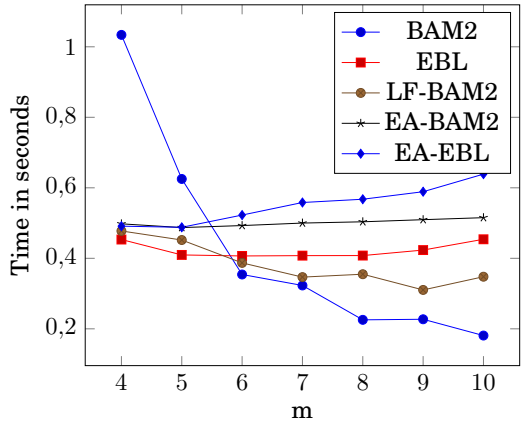


Figure 9.20. Execution times of algorithms (in seconds) for exact JPM for protein data on Haswell.

Fig. 9.19 and 9.20 show the result of execution times for exact algorithms for English and protein data respectively on Haswell. For English

data, LF-BAM2 is a clear winner for all the pattern lengths except for $m = 8$ and 10 where BAM2 performs the best. For protein data, EBL is the fastest for shorter patterns and BAM2 performs the best for patterns larger than 5.

10. Conclusion and Future Scope

We have developed several online algorithms for exact and approximate jumbled matching using backward and forward methods. In some of our algorithms, we have used the shared bin technique in order to be able to handle longer patterns in a large alphabet. In addition to that, we also applied the SIMD (Single Instruction Multiple Data) approach to exact jumbled matching. The techniques of shared bins and SIMD showed to be useful for jumbled matching.

Our algorithms are an outcome of a long series of experimentation. We carried out extensive practical experiments to compare the new algorithms with earlier algorithms. In our tests, we used English, DNA, protein and binary text with a wide range of pattern lengths. The new algorithms showed to be competitive in most scenarios. In almost every tested case, the best of our algorithms are faster than any previous solution, and in many cases the speed of the best algorithm doubles the speed of previous solutions.

Our algorithms include extensions of the BAM algorithm for both exact and approximate jumbled matching by making use of various approaches. One of the algorithms applies backward matching of 2-grams as in the SBNDM2 algorithm. We also developed specialized algorithms for binary data for both exact and approximate case.

For approximate jumbled matching, in the case of a larger alphabet and the error count 1 and 2, ABAM and AF obtain the best execution times achieving speedup of two or more in some cases. However for the error value 3, AF outplays all other algorithms. For smaller alphabets (except the binary alphabet), AF also achieves the best execution time.

All forward algorithms are clearly linear. The speed of their approximate versions does not depend on the value of k. The experiments support the assertion that the backward algorithms are sublinear on average for small

k and large m.

We also introduced the SIMD approach to jumbled matching problem in order to quickly filter the text. One of our two SIMD algorithms uses the equal any operation and the other searches for the least frequent character of the pattern. Our experiments show that in most of the cases, our best SIMD algorithm is 30% faster than other previous algorithms for short English patterns.

Furthermore, we have presented a novel algorithm for episode matching that finds the maximal frequent episodes from the given sequence using bit parallel approach.

We believe that there is still room to improve our results, such as more sophisticated character selection for shared bins, which may lead to the faster solutions. It should be realized that if the latency of the used SIMD instructions would improve in future processors, the running times of the SIMD algorithms will change respectively.

Bibliography

[1] A. AMIR, A. APOSTOLICO, T. HIRST, G. M. LANDAU, N. LEWENSTEIN, L. ROZENBERG: Algorithms for jumbled indexing, jumbled border and jumbled square on run–length encoded strings. In String Processing and Information Retrieval – 21st International Symposium, SPIRE 2014, Ouro Preto, Brazil, October 20–22, 2014, Proceedings, volume 8799 of Lecture Notes in Computer Science, pages 45–51. Springer, 2014.

[2] D. BELAZZOUGUI, A. PIERROT, M. RAFFINOT, S. 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, volume 8283 of Lecture Notes in Computer Science, pages 66–77. Springer, 2013.

[3] G. BENSON: Composition alignment. In Algorithms in Bioinformatics – 3rd International Workshop, WABI 2003, Budapest, Hungary, September 15–20, 2003, Proceedings, volume 2812 of Lecture Notes in Computer Science, pages 447–461. Springer, 2003.

[4] S. BÖCKER: Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt. Journal of Computational Biology 1(6):1110–1134, 2008.

[5] S. BÖCKER: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. Bioinformatics 23(2):5–12, 2007.

[6] S. BÖCKER, K. JAHN, J. MIXTACKI, J. STOYE: Computation of median gene clusters. Journal of Computational Biology 16(8):1085–1099, 2009.

[7] R. S. BOYER, J. S. MOORE: A fast string searching algorithm. Commun. of the ACM 20(10):762–772, 1977.

[8] P. BURCSI, F. CICALESE, G. FICI, ZS. LIPTÁK: Algorithms for jumbled pattern matching in strings. Int. J. Found. Comput. Sci. 23(2):357–374, 2012.

[9] P. BURCSI, F. CICALESE, G. FICI, ZS. LIPTÁK: On approximate jumbled pattern matching in strings. Theory Comput. Syst. 50(1):35–51, 2012.

[10] P. BURCSI, F. CICALESE, G. FICI, ZS. LIPTÁK: On table arrangement, scrabble freaks, and jumbled pattern matching. In Fun with Algorithms – 5th International Conference, FUN 2010, Ischia, Italy, June 2–4, 2010, Proceedings, volume 6099 of Lecture Notes in Computer Science, pages 89–101. Springer, 2010.

[11] D. CANTONE, S. FARO: Efficient online Abelian pattern matching in strings by simulating reactive multi–automata. In Prague Stringology Conference, PSC 2014, Prague, Czech Republic, September 1–3, 2014, Proceedings, pages 30–42, 2014.

[12] I. CASTELLANOS, Y. PINZÓN: Efficient algorithm for δ–Approximate Jumbled Pattern Matching. In Prague Stringology Conference, PSC 2015, Prague, Czech Republic, August 24–26, 2015, Proceedings, pages 47–56, 2015.

[13] C. CHAUVE, Y. DIEKMANN, S. HEBER, J. MIXTACKI, S. RAHMANN, J. STOYE: On Common Intervals with Errors. Technical report, Bielefeld University, 2006.

[14] F. CICALESE, G. FICI, ZS. LIPTAK: Searching for jumbled patterns in strings. In Prague Stringology Conference, PSC 2009, Prague, Czech Republic, August 31 – September 2, 2009, Proceedings, pages 105–117, 2009.

[15] T. CHHABRA, S. S. GHUMAN, J. TARHIO: Tuning algorithms for jumbled matching. In Prague Stringology Conference, PSC 2015, Prague, Czech Republic, August 24–26, 2015, Proceedings, pages 57–66, 2015.

[16] M. CHIMANI, K. KLEIN: Algorithm engineering: Concepts and practice. In Experimental Methods for The Analysis of Optimization Algorithms, pages 131–158. Springer, 2010.

[17] S. CHO, J. C. NA, K. PARK, J. S. 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.

[18] E. DOMANN, T. HAIN, R. GHAI, A. BILLION, C. KUENNE, K. ZIMMERMANN, T. CHAKRABORTY: Comparative genomic analysis for the presence of potential enterococcal virulence factors in the probiotic Enterococcus faecalis strain Symbioflor 1. International Journal of Medical Microbiology, 297(7–8):533–539, 2007. Special issue: Pathogenomics.

[19] B. ĎURIAN, J. HOLUB, H. PELTOLA, J. TARHIO: Improving practical exact string matching. Inf. Process. Lett. 110(4):148–152, 2010.

[20] E. EJAZ: Abelian Pattern Matching in Strings. Ph.D. Thesis, Dortmund University of Technology(2010), http://d-nb.info/1007019956.

[21] R. ERES, G. M. LANDAU, L. PARIDA: Permutation pattern discovery in biosequences. Journal of Computational Biology 11(6):1050–1060, 2004.

[22] S. FARO: Private Communication.

[23] S. FARO, M. O. KULEKCI: Fast packed string matching for short patterns. In 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013, Proceedings, pages 113–121, 2013.

[24] S. FARO, M. O. KULEKCI: Fast and flexible packed string matching. J. Discrete Algorithms 28:61–72, 2014.

[25] S. FARO, T. LEQROC: Smart: string matching algorithms research tool (2015), http://www.dmi.unict.it/∼faro/smart/

[26] T. GAGIE, D. HERMELIN, G. M. LANDAU, O. WEIMANN: Binary jumbled pattern matching on trees and tree-like structures. Algorithmica 73(3):571–588, 2015.

[27] S. S. GHUMAN, J. TARHIO: Jumbled matching with SIMD. In Prague Stringology Conference, PSC 2016, Prague, Czech Republic, August 29-31, 2016, Proceedings, pages 114–124, 2016.

[28] E. GIAQUINTA, S. GRABOWSKI: New algorithms for binary jumbled pattern matching. Inf. Process. Lett. 113(14):538–542, 2013.

[29] S. GRABOWSKI, S. FARO, E. GIAQUINTA: String matching with inversions and translocations in linear average time (most of the time). Inf. Process. Lett. 111(11):516–520, 2011.

[30] R. GROSSI, F. LUCCIO: Simple and efficient string matching with k mismatches. Inf. Process. Lett. 33(3):113–120, 1989.

[31] R. GROSSI, A. GUPTA, J. S. VITTER: High-order entropy-compressed text indexes. In 14th Annual ACM–SIAM Symposium on Discrete Algorithms, Baltimore, Maryland, USA, January 12–14, 2003, Proceedings, pages 841–850, 2003.

[32] M. HASSABALLAH, S. OMRAN, Y. B. MAHDY: A review of SIMD multimedia extensions and their usage in scientific and engineering applications. Comput. J. 51(6):630–649, 2008.

[33] S. HEBER, J. STOYE: Algorithms for finding gene clusters. In Algorithms in Bioinformatics - 1st International Workshop, WABI 2001, Aarhus, Denmark, August 28–31, 2001, Proceedings, volume 2149 of Lecture Notes in Computer Science, pages 252—263. Springer, 2001.

[34] S. HEBER, R. MAYR, J. STOYE: Common intervals of multiple permutations. Algorithmica 60(2):175–206, 2009.

[35] T. HIRVOLA: Private Communication.

[36] A. HUME, D. SUNDAY: Fast string searching. Software – Practice and Experience 21(11):1221–1248, 1991.

[37] 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).

[38] INTEL: http://ark.intel.com/products/codename/29896/Lynnfield.

[39] INTEL: The Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.

[40] H. JEONG, S. KIM, W. LEE, S. H. MYUNG : Performance of SSE and AVX instruction sets. CoRR, abs/1211.0820, 2012.

[41] P. JOKINEN, J. TARHIO, E. UKKONEN: A comparison of approximate string matching algorithms. Software – Practice and Experience 26(12):1439–1458, 1996.

[42] S. KARLIN: Detecting anomalous gene clusters and pathogenicity islands in diverse bacterial genomes. Trends in Microbiology 9(7):335–343, 2001.

[43] J. KIM, P. EADES, R. FLEISCHER, S. H. HONG, C. S. ILIOPOULOS, K. PARK, S. J. PUGLISI, T. TOKUYAMA: Order- preserving matching. Theor. Comput. Sci. 525:68–79, 2014.

[44] T. KOCIUMAKA, J. RADOSZEWSKI, W. RYTTER: Efficient Indexes for Jumbled Pattern Matching with Constant-Sized Alphabet. In Algorithms – 21st Annual European Symposium, Sophia Antipolis, France, September 2–4, 2013, Proceedings, volume 8125 of Lecture Notes in Computer Science, pages 625–636. Springer, 2013.

[45] M. O. KÜLEKCI: Filter based fast matching of long patterns by using SIMD instructions. In Prague Stringology Conference, PSC 2009, Prague, Czech Republic, August 31 – September 2, 2009, Proceedings, pages 118–128, 2009.

[46] S. LADRA, O. PEDREIRA, J. DUATO, N. 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, volume 7503 of Lecture Notes in Computer Science, pages 254–267. Springer, 2012.

[47] H. MANNILA, H. TOIVONEN, A. I. VERKAMO: Discovery of frequent episodes in event sequences. Data Min. Knowl. Discov. 1(3):259–289,1997.

[48] E. M. MARCOTTE, M. PELLEGRINI, H. L. NG, D. W. RICE, T.O. YEATES, D. EISENBERG: Detecting protein function and protein-protein interactions from genome sequences. Science 285(5428):751–753, 1999.

[49] T. M. MOOSA, M. S. RAHMAN: Indexing permutations for binary strings. Inf. Process. Lett. 110(18–19):795–798, 2010.

[50] G. NAVARRO, M. RAFFINOT: A Bit-parallel Approach to Suffix Automata: Fast Extended String Matching. In Combinatorial Pattern

Matching - 9th Annual Symposium, CPM 1998, Piscataway, New Jersey, USA, July 20–22, 1998, Proceedings, volume 1448 of Lecture Notes in Computer Science, pages 254–267. Springer, 1998.

[51] G. NAVARRO, M. RAFFINOT: Fast and Flexible String Matching by Combining Bitparallelism and Suffix automata. ACM Journal of Experimental Algorithmics 5(4):1–36, 2000.

[52] G. NAVARRO: Multiple approximate string matching by counting. In WSP 1997, 4th South American Workshop on String Processing, 2011, pages 95–111.

[53] G. NAVARRO, M. RAFFINOT: Flexible Pattern Matching in Strings. Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, New York 2002.

[54] R. OVERBEEK, M. FONSTEIN, M. DSOUZA, G.D. PUSCH, N. MALYSEV: Use of contiguity on the chromosome to predict functional coupling. Silico Biology 1(2):93–108, 1998.

[55] R. OVERBEEK, M. FONSTEIN, M. D'SOUZA, G.D. PUSCH, N. MALTSEV: The Use of Gene Clusters to Infer Functional Coupling. Proceedings of the National Academy of Sciences 96(6):2896-–2901, 1999.

[56] H. PELTOLA, J. TARHIO: Alternative Algorithms for Bit-Parallel String Matching, 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.

[57] A. SALOMAA: Counting (scattered) subwords. Bulletin of the EATCS, 81:165–179, 2003.

[58] P. SANDERS: Algorithm engineering – an attempt at a definition using sorting as an example. In 12th Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010, Proceedings, pages 55–61, 2010.

[59] T. SCHMIDT, J. STOYE: Quadratic time algorithms for finding common intervals in two and more sequences. In Combinatorial Pattern Matching – 15th Annual Symposium, CPM 2004, Istanbul,Turkey, July 5–7, 2004, Proceedings, volume 3109 of Lecture Notes in Computer Science, pages 347–358. Springer, 2003.

[60] H. TOIVONEN: Discovery of Frequent Patterns in Large Data Collections. Ph. D. dissertation, University of Helsinki, 1996.

[61] T. UNO, M. YAGIURA: Fast algorithms to enumerate all common intervals of two permutations. Algorithmica 26(2):290–309, 2000.

[62] A. WIEZER, R. MERKL: A comparative categorization of gene flux in diverse microbial species. Genomics 86(4):462–475, 2005.

[63] S. YOSSI: Fast canonization of circular strings. Journal of Algorithms 2(2):107–121, 1981.

In Computer Science, the problem of finding the occurrences of a given string is a common task. There are many different variations of the problem. We consider the problem of jumbled pattern matching (JPM) (also known as Abelian pattern matching or permutation matching) where the objective is to find all permuted occurrences of a pattern in a text. Jumbled pattern matching has numerous applications in the field of bioinformatics. For instance, jumbled matching can be used to find those genes that are closely related to one another. Besides exact jumbled matching we study approximate jumbled matching where each occurrence is allowed to contain at most k wrong or superfluous characters. We present online algorithms applying bitparallelism to both types of jumbled matching. Two of our algorithms are filtration methods applying SIMD (Single Instruction Multiple Data) computation. Furthermore, we have developed a bitparallel algorithm for episode matching. This algorithm finds the maximal parallel episodes of a given sequence. Most of the other new algorithms are variations of earlier methods.

ISBN 978-952-60-7757-4 (printed)
ISBN 978-952-60-7758-1 (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

Aalto-DD 242/2017