TKK Research Reports in Computer Science and Engineering A TKK-CSE-A1/09
Espoo 2009

IMPROVED ALGORITHMS FOR STRING SEARCHING PROBLEMS

Doctoral Dissertation

Leena Salmela

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 1st of June, 2009, at 12 noon.

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Computer Science and Engineering
P.O. Box 5400
FI-02015 TKK
FINLAND
URL: http://www.cse.tkk.fi/
Tel. +358 9 451 3228
Fax +358 9 451 3293
e-mail: lsalmela@cs.hut.fi

ISBN 978-951-22-9887-7
ISBN 978-951-22-9888-4 (PDF)
ISSN 1797-6928
ISSN 1797-6936 (PDF)
URL: http://lib.tkk.fi/Diss/2009/isbn9789512298884/

Multiprint Oy
Espoo 2009

ABSTRACT OF DOCTORAL DISSERTATION	HELSINKI UNIVERSITY OF TECHNOLOGY P. O. BOX 1000, FI-02015 TKK http://www.tkk.fi/

Author Leena Salmela

Name of the dissertation

Improved Algorithms for String Searching Problems

Manuscript submitted 09.02.2009	Manuscript revised 11.05.2009
Date of the defence 01.06.2009	
[X] Monograph	[ ] Article dissertation (summary + original articles)

Faculty	Faculty of Information and Natural Sciences
Department	Department of Computer Science and Engineering
Field of research	Software Systems
Opponent(s)	Prof. Maxime Crochemore
Supervisor	Prof. Jorma Tarhio
Instructor	Prof. Jorma Tarhio

Abstract

We present improved practically efficient algorithms for several string searching problems, where we search for a short string called the pattern in a longer string called the text. We are mainly interested in the online problem, where the text is not preprocessed, but we also present a light indexing approach to speed up exact searching of a single pattern. The new algorithms can be applied e.g. to many problems in bioinformatics and other content scanning and filtering problems.

In addition to exact string matching, we develop algorithms for several other variations of the string matching problem. We study algorithms for approximate string matching, where a limited number of errors is allowed in the occurrences of the pattern, and parameterized string matching, where a substring of the text matches the pattern if the characters of the substring can be renamed in such a way that the renamed substring matches the pattern exactly. We also consider searching multiple patterns simultaneously and searching weighted patterns, where the weight of a character at a given position reflects the probability of that character occurring at that position.

Many of the new algorithms use the backward matching principle, where the characters of the text that are aligned with the pattern are read backward, i.e. from right to left. Another common characteristic of the new algorithms is the use of q-grams, i.e. q consecutive characters are handled as a single character. Many of the new algorithms are bit parallel, i.e. they pack several variables to a single computer word and update all these variables with a single instruction.

We show that the q-gram backward string matching algorithms that solve the exact, approximate, or multiple string matching problems are optimal on average. We also show that the q-gram backward string matching algorithm for the parameterized string matching problem is sublinear on average for a class of moderately repetitive patterns. All the presented algorithms are also shown to be fast in practice when compared to earlier algorithms.

We also propose an alphabet sampling technique to speed up exact string matching. We choose a subset of the alphabet and select the corresponding subsequence of the text. String matching is then performed on this reduced subsequence and the found matches are verified in the original text. We show how to choose the sampled alphabet optimally and show that the technique speeds up string matching especially for moderate to long patterns.

Keywords	string matching, approximate string matching, multiple string matching, parameterized string matching, weighted string matching, q-grams, bit parallelism, text indexing
ISBN (printed) 978-951-22-9887-7	ISSN (printed) 1797-6928
ISBN (pdf) 978-951-22-9888-4	ISSN (pdf) 1797-6936
Language English	Number of pages 153 p.
Publisher	Department of Computer Science and Engineering
Print distribution	Department of Computer Science and Engineering
[X] The dissertation can be read at http://lib.tkk.fi/Diss/2009/isbn9789512298884/	

VÄITÖSKIRJAN TIIVISTELMÄ	TEKNILLINEN KORKEAKOULU PL 1000, 02015 TKK http://www.tkk.fi/

Tekijä Leena Salmela

Väitöskirjan nimi

Parannettuja algoritmeja merkkijonohakuongelmiin

Käsikirjoituksen päivämäärä 09.02.2009	Korjatun käsikirjoituksen päivämäärä 11.05.2009
Väitöstilaisuuden ajankohta 01.06.2009	
☒ Monografia	☐ Yhdistelmäväitöskirja (yhteenveto + erillisartikkelit)

Tiedekunta	Informaatio- ja luonnontieteiden tiedekunta
Laitos	Tietotekniikan laitos
Tutkimusala	Ohjelmistojärjestelmät
Vastaväittäjä(t)	Prof. Maxime Crochemore
Työn valvoja	Prof. Jorma Tarhio
Työn ohjaaja	Prof. Jorma Tarhio

Tiivistelmä

Esitämme parannettuja käytännössä tehokkaita algoritmeja useisiin merkkijonohakuongelmiin, joissa etsitään lyhyttä merkkijonoa eli hahmoa pitkästä merkkijonosta eli tekstistä. Keskitymme pääasiassa ongelman muunnelmaan, missä tekstiä ei esikäsitellä, mutta esitämme myös kevyen hakemistorakenteen, joka nopeuttaa yhden hahmon tarkkaa hakua. Esitettyjä uusia algoritmeja voidaan soveltaa mm. moniin bioinformatiikan ongelmiin ja erilaisiin haku- ja suodatusongelmiin.

Tarkan merkkijonohaun lisäksi kehitämme algoritmeja moniin muihin merkkijonohakuongelmiin. Käsittelemme merkkijonojen likimääräistä hakua, missä sallitaan rajattu määrä virheitä hahmon esiintymissä, ja parametrisoitua hakua, missä tekstin osajono täsmää hahmoon, jos osajonon merkit voidaan nimetä uudelleen siten, että hahmo täsmää tarkasti tähän uudelleennimettyyn osajonoon. Tarkastelemme myös usean hahmon yhtäaikaista hakua ja painotettujen hahmojen hakua, missä kunkin merkin paino kussakin positiossa kuvaa kyseisen merkin todennäköisyyttä esiintyä kyseisessä positiossa.

Monet uusista algoritmeista lukevat hahmon kanssa kohdistetun tekstin osajonon taaksepäin eli oikealta vasemmalle. Toinen yhteinen piirre esitetyille algoritmeille on q-piirteiden käyttö eli algoritmit käsittelevät q:ta peräkkäistä merkkiä yhtenä merkkinä. Monet näistä uusista algoritmeista ovat bittirinnakkaisia eli ne pakkaavat monta muuttujaa samaan tietokoneen sanaan ja päivittävät kaikkia näitä muuttujia yhdellä käskyllä.

Näytämme, että taaksepäin täsmäävät q-piirrealgoritmit, jotka ratkaisevat tarkan, likimääräisen tai usean hahmon merkkijonohaun, ovat keskimäärin optimaalisia. Lisäksi todistamme, että parametrisoidun haun taaksepäin täsmäävä q-piirrealgoritmi on keskimäärin alilineaarinen joukolle kohtalaisen toisteisia hahmoja. Näytämme myös, että kaikki esitetyt algoritmit ovat käytännössä nopeita verrattuna aikaisempiin algoritmeihin.

Lopuksi esitämme aakkostonkarsintamenetelmän, joka nopeuttaa tarkkaa merkkijonohakua. Menetelmässä valitaan aakkoston osajoukko ja vastaava tekstin alijono. Hahmoa haetaan tästä lyhennetystä tekstistä ja löydetyt esiintymät tarkistetaan alkuperäisestä tekstistä. Näytämme, miten aakkoston osajoukko valitaan optimaalisesti ja että menetelmä nopeuttaa merkkijonohakua erityisesti kohtalaisen pitkillä hahmoilla.

Asiasanat merkkijonohaku, likimääräinen merkkijonohaku, monen hahmon haku, parametrisoitu merkkijonohaku, painotettujen hahmojen haku, q-piirre, bittirinnakkaisuus, tekstin indeksointi

ISBN (painettu) 978-951-22-9887-7	ISSN (painettu) 1797-6928
ISBN (pdf) 978-951-22-9888-4	ISSN (pdf) 1797-6936
Kieli Englanti	Sivumäärä 153 s.

Julkaisija Tietotekniikan laitos

Painetun väitöskirjan jakelu Tietotekniikan laitos

☒ Luettavissa verkossa osoitteessa http://lib.tkk.fi/Diss/2009/isbn9789512298884/

Preface

First of all, I would like to thank my supervisor, Professor Jorma Tarhio. He recruited me to the String Algorithms Group when I was still an undergraduate student and introduced me to the many interesting problems in string algorithms. I would also like to thank Hannu Peltola who has always been ready to listen to my complaints and questions and to help with the more practical mysteries of university life. Also other members of SAG deserve their thanks. It has been a pleasure to work with you.

This thesis was written while I was working at the Department of Computer Science and Engineering at Helsinki University of Technology. For financial support I would like to thank Helsinki Graduate School in Computer Science and Engineering and the Academy of Finland.

I am also grateful for the opportunity to visit University of Chile in April 2008. It was very inspiring to work with Professor Gonzalo Navarro and Francisco Claude. The two weeks away from the usual distractions also proved to be very helpful in getting started on writing this thesis.

I would also like to thank the pre-examiners, Professor Esko Ukkonen and Professor Erkki Sutinen, for many useful comments that helped to improve this work.

Finally, I would like to thank my family and friends for their support throughout my studies. Especially, I thank Teemu Takanen for taking the cover photo for this thesis.

Espoo, May 2009

Leena Salmela

Contents

Chapter 1

Introduction

The most fundamental problem in string algorithms is the exact string matching problem. The input of this problem is two strings, a text and a pattern, and the task is to find all exact occurrences of the pattern in the text. Over the years, several variations of this basic problem have emerged. In the approximate string matching problem, a limited number of errors is allowed in the occurrences of the pattern in the text. Another variation is the parameterized string matching problem [16], where the pattern matches a substring of the text if the characters of the text substring can be renamed in such a way that the pattern matches the renamed substring exactly. Other variations include searching for multiple patterns simultaneously and searching for a weighted pattern, where in each position of the pattern a weight is given to each character of the alphabet describing the probability of the character occurring at that position.

Algorithms that solve string matching problems come in two flavors: online and indexing. Online algorithms can preprocess the pattern, but they do not preprocess the text. Indexing algorithms are able to speed up searching by preprocessing the text. This work concentrates mainly on online algorithms, but in Chapter 7, we will also look at a light weight indexing approach to speed up online searching.

Lower bound on the worst case complexity of the online approach to the exact string matching problem has been proved to be $\Omega(n)$, where n is the length of the text. The first algorithm to reach this bound was the Knuth-Morris-Pratt algorithm [57]. In practice, the best algorithms do not inspect every character of the text, and the lower bound of the average case complexity of the problem has been proved to be $\Omega(n \log_\sigma m/m)$ [111], where σ is the size of the alphabet and m the length of the pattern. For example, the Backward DAWG Matching (BDM) algorithm [30] has been proved to be optimal on average, but other nonoptimal sublinear algorithms, like the Boyer-Moore-Horspool algorithm [49], are very competitive in practice. Similar results have been shown for the approximate string matching problem [25] and the multiple string matching problem and a combination of them [42]. In this work, the emphasis is on developing practical algorithms with good average case complexity for several variations of the string matching problem.

In general, sublinear string matching algorithms work best when the alphabet is large and the distribution of characters is even because then the probability of matching a random string of characters is low. As this is not the case in many practical situations, like searching natural language texts or DNA sequences, we show that practical methods can be developed to boost string matching algorithms in these cases.

1.1 Applications

1.1.1 Bioinformatics

DNA and protein sequences have a central role in modern biology. The rapidly growing databases of such sequences present a challenge for developing efficient string matching algorithms. The DNA sequences of related species and even individuals within a species can differ slightly, and thus there is a need for approximate searching of the sequences in addition to exact searching.

In many cases, approximate matching is not sufficient to model the complex biological variation present in real sequences. A weighted pattern is one model that has been successfully applied to model, for example, transcription factor binding sites [93] and protein families [44]. In bioinformatics, the terms position weight matrices, position specific scoring matrices, or profiles are often used to refer to weighted patterns.

New DNA sequencers produce massive amounts of short reads of DNA text in a single run [20]. If a reference genome is known, a first step in processing these short reads is to map them to the reference genome. As the number of these short sequences is very large, new efficient multiple string matching algorithms are needed to complete this task.

1.1.2 Data Scanning

Multiple string matching algorithms are needed in various data scanning problems. Two examples of such an application are anti-virus scanning [74] and intrusion detection [37, 68, 102].

In anti-virus scanning, signatures are defined to describe known computer viruses, and the first task in these applications is to locate these signatures in large amounts of data. When a signature is found, more sophisticated methods are needed to confirm the presence of a computer virus. The rapidly growing set of signatures calls for efficient multiple string matching algorithms.

In intrusion detection applications, strings related to attacks are defined. These strings are then searched for in network traffic, and the system is alerted for further inspection if a suspicious sequence of these strings is found.

1.1.3 Plagiarism Detection

Plagiarism has become a growing concern in education [55, 92]. In computer science, a particular problem is the copying of code for programming assignments. A common modification to a copied program is the change of variable names. If the program is considered as a sequence of tokens, parameterized matching can detect a copied program even if variable names have been changed [41].

1.1.4 Image Searching

Searching for images is an extension of string matching to two-dimensional objects. Several algorithms have been presented to solve the exact matching problem [15, 19, 22, 43, 53, 100, 114]. We consider the two-dimensional version of parameterized string matching, which can identify an image even if its color map has been changed.

1.2 Results and Contributions

The main results of this thesis are as follows:

- We show that the average case complexity of the Boyer-Moore-Horspool algorithm with q-grams is $\mathcal{O}(n \log_\sigma m/m)$, which is optimal.
- We present practical Boyer-Moore-Horspool style algorithms for approximate string matching with optimal average case complexity of $\mathcal{O}(n(\log_\sigma m + k)/m)$ for $k/m < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$, where k is the maximum number of differences in the occurrences of the pattern in the text.
- We develop Boyer-Moore-Horspool style algorithms for parameterized string matching in one and two dimensions with sublinear average case complexity for moderately repetitive patterns.
- Two algorithms for multiple pattern matching with optimal average case complexity of $\mathcal{O}(n \log_\sigma(rm)/m)$, where r is the number of patterns, and another algorithm with linear average case complexity are presented. The algorithms are practical even for very large pattern sets.
- Practical algorithms for the weighted string matching problem are developed.
- A light weight indexing scheme to speed up online search on natural language texts is presented.

This thesis includes material from the original publications [26, 52, 60, 86, 87, 88, 89, 90], but for example, many of the analytical results are new. The co-authors have a contribution in some of the results. Most notably, the initial ideas for the multiple string matching algorithms in Chapter 5 are by Jorma Tarhio and Jari Kytöjoki, and

the initial idea for the sampled semi-index in Chapter 7 is by Gonzalo Navarro. Most of the algorithms have been implemented by the author. The sampled semi-index in Chapter 7 was implemented by Hannu Peltola and the approximate matching algorithms in Chapter 3 were developed jointly by the author, Janne Auvinen, Petri Kalsi, and Jorma Tarhio.

1.3 Organization

We start by introducing the needed definitions, basic algorithms, and some tools for analyzing the new algorithms in Chapter 2. We then study two variations of string matching where the criteria for matching have been changed. Chapter 3 studies the approximate string matching problem with an emphasis on small alphabets, and Chapter 4 discusses efficient algorithms for the parameterized matching problem. Then we turn to two variations of string matching where the pattern is more complex. Chapter 5 studies multiple string matching with an emphasis on very large pattern sets, and Chapter 6 explores the weighted string matching problem. In Chapter 7, we return to the exact string matching problem in the context of natural language texts and develop an alphabet sampling technique to speed up the search process.

Chapter 2

Background

2.1 Basic Concepts

2.1.1 Alphabets, Strings, and String Matching

Definition 2.1. *An* alphabet Σ *is a set of* characters*. The size of the alphabet is denoted by* σ*. An* integer alphabet *is a set of integers from the range* $[1, \sigma]$*. A* constant alphabet *is a finite set of constant size.*

Most of the algorithms presented in this thesis assume that the alphabet is an integer alphabet. A constant alphabet is easily transformed to an integer alphabet by preparing a mapping table that maps each character to a unique integer in the range $[1, \sigma]$.

Definition 2.2. *A* string *is a sequence of characters drawn from an alphabet. If* $S = s_1 s_2 \dots s_n$ *is a string, then* $S' = s_{i_1} s_{i_2} \dots s_{i_m}$*, where* $1 \leq i_1 < i_2 < \dots < i_m \leq n$*, is a* subsequence *of* S*. Furthermore,* $S'' = s_i s_{i+1} \dots s_j$*, where* $1 \leq i \leq j \leq n$*, is a* substring *of* S*. If* $i = 1$*, then* S'' *is a* prefix *of* S*, and if* $j = n$*, then* S'' *is a* suffix *of* S*. The empty string* ε *of length* 0 *is both a prefix and a suffix of any string.*

We will use capital letters to denote strings and the corresponding lower case letters to denote the characters of the string. We will denote by Σ^q the set of all strings of length q drawn from an alphabet Σ.

The problems studied in this thesis are string matching problems. The simplest string matching problem is the exact string matching problem.

Problem 2.3. *Given two strings, a text* $T = t_1 \dots t_n$ *and a pattern* $P = p_1 \dots p_m$*, the* exact string matching problem *is to find all substrings of the text that match the pattern. These matching substrings are called the* occurrences *of the pattern.*

There are many variations of this basic problem. The criteria for matching can be different. For example, a limited number of substitutions, insertions, or deletions can be allowed. The pattern can also be a more complex structure, like a set of strings. These variations will be defined in the forthcoming chapters as they are needed.

2.1.2 Bit Vectors

Definition 2.4. *A* bit vector *is a sequence of bits. We will denote a bit vector of width w as $E = e_w \dots e_1$, where e_1 is the least significant bit, and e_w is the most significant bit.*

We define the following operators on bit vectors: $|$ denotes the bit-wise or operator, $\&$ the bit-wise and, and $\hat{}$ the bitwise xor operator. The operation $E \ll n$ shifts the bits of the bit vector E n positions to the left inserting zeroes to the least significant bits, and $\gg$ shifts the bits to the right in a similar fashion. Arithmetic operations, like addition, are defined on bit vectors as with normal binary numbers. We use the shorthands 1^x and 0^x to denote a bit value that is repeated x times.

2.2 Common Algorithmic Techniques

2.2.1 q-Grams

Many string matching algorithms rely on a fairly large alphabet for good performance. The idea behind using q-grams is to make the alphabet perceived by the algorithm larger. When using q-grams, we process q consecutive characters as a single character. There are two ways of transforming a string of characters into a string of q-grams. We can either use overlapping q-grams or consecutive q-grams. When overlapping q-grams are used, a q-gram starts at every position of the original text, while with consecutive q-grams, a q-gram starts in every q:th position. For example, transforming the word "pony" into overlapping 2-grams results in the string "po-on-ny", and transforming it into consecutive 2-grams yields the string "po-ny".

In many algorithms, q-grams are used to index tables. For maximum performance, it is crucial how this index value of a q-gram is computed. One way is to map the characters to integers and to use the following loop to construct a bit representation of a q-gram:

$$\begin{aligned}&\mathrm{bits} = 0\\&\text{for } (i = 1 \text{ to } q)\\&\quad \mathrm{bits} = (\mathrm{bits} \ll b) \mid \mathrm{map}(\mathrm{gram}[i]) \ ,\end{aligned}$$

where gram is the textual representation of the q-gram, map is an inline function that maps the characters to integers, b is the number of bits needed to represent the integers, and bits is the bit representation of the q-gram. When using bytes as characters, 2-grams and 4-grams can be easily read by a single instruction on machines that do not require memory references to halfwords or words to be aligned on (half)word boundaries [38].

2.2.2 Bit Parallelism

Bit parallelism takes advantage of the bit operations of processors by packing several variables into a single computer word. These variables can then be updated in a single instruction making use of the intrinsic parallelism of bit operations. For example, if we needed to keep track of $m \leq w$ boolean variables, where w is the length of the computer word, we could store all these variables in a single computer word. Furthermore, we can update all the variables in one instruction instead of m instructions. As the length of the computer word in modern processors is 32 or 64, this technique can give us a significant speedup.

2.3 String Matching Algorithms

A myriad of string matching algorithms have been developed. Here we will review only those algorithms that are used when constructing the new algorithms presented in this thesis. For more information, see the many books on string matching algorithms [32, 33, 46, 80]. All the following algorithms solve the exact string matching problem for a single pattern.

2.3.1 Boyer-Moore-Horspool Algorithm

The Boyer-Moore algorithm [23] was the first sublinear string matching algorithm. The algorithm processes the text in windows of length m. The key idea of the algorithm is that in each window the characters are read from right to left, and when a mismatch is found, the window is shifted based on the text characters read. In many cases, this allows the algorithm to entirely skip reading some text characters. The original algorithm uses two shifting heuristics, the bad character heuristic and the good suffix rule. The bad character heuristic determines the shortest possible shift such that the rightmost character of the current window matches the pattern after the shift. If no such shift is possible (i.e. the rightmost character of the current window does not occur in the pattern), the bad character heuristic recommends a shift of length m. The good suffix rule is more involved. It assures that the matching suffix of the current window matches the pattern also after the shift if it is then aligned with the pattern.

Horspool [49] proposed to use only the bad character heuristic because in most cases that heuristic determines the shift length. In practice, the Boyer-Moore-Horspool algorithm is faster than the original Boyer-Moore algorithm. Several other improvements to the Boyer-Moore algorithm have also been proposed [7, 21, 30, 50, 85, 96, 98, 113].

The preprocessing phase of the Boyer-Moore-Horspool algorithm consists of calculating the bad character function $S[c]$, which will be used for shifting the window during the search phase. The bad character function is defined as the distance from the end of the pattern $P = p_1 p_2 \ldots p_m$ to the last occurrence of the character c in the

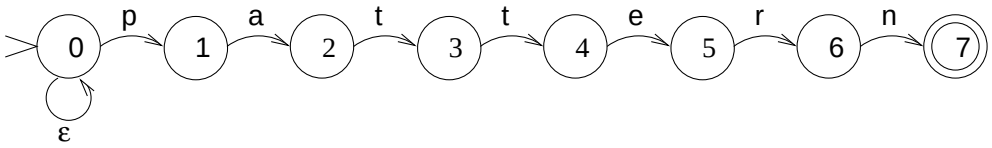


Figure 2.1: The nondeterministic automaton recognizing the pattern "pattern"

pattern excluding the last character:

$$S[c] = \min\{h \mid p_{m-h} = c, 1 \leq h \leq m-1\} \ .$$

If the character c does not appear in the prefix of the pattern $p_1 \ldots p_{m-1}$, $S[c] = m$.

In the searching phase, the last character of the window is compared with the last character of the pattern. If they match, the whole window is compared against the pattern to check for a match. After that or if the last characters did not match, the window is shifted by $S[c]$, where c is the last character of the window. The worst case complexity of the Boyer-Moore-Horspool algorithm is $\mathcal{O}(mn)$, and the average case complexity is $\mathcal{O}(n(1/m + 1/\sigma))$ [10].

It is well known (see e.g. [9, 23, 57]) that the use of q-grams can increase the average length of shift in the algorithms of Boyer-Moore type. In the Boyer-Moore-Horspool algorithm, the bad character function is now replaced by the bad q-gram function, which is defined as the distance from the end of the pattern to the last occurrence of the q-gram G excluding the last q-gram of the pattern:

$$S_q[G] = \min\{h \mid p_{m-h-q+1} \ldots p_{m-h} = G, 1 \leq h \leq m-q\} \ .$$

If the q-gram does not occur in the prefix of the pattern $p_1 \ldots p_{m-1}$, $S_q[G] = m-q+1$. In the searching phase, the shift is then based on the last q-gram of the text window. This basic bad q-gram function can be improved by defining the maximal shift length to be m and also considering the cases when the suffix of the q-gram matches the prefix of the pattern when defining the function.

2.3.2 Shift-Or Algorithm

Shift-and [2, 34] was the first bit parallel string matching algorithm, but shift-or [11] is a very similar algorithm, which can be implemented more efficiently. The shift-or algorithm is a bit-parallel algorithm simulating a simple nondeterministic automaton that recognizes the pattern. An example of such an automaton is shown in Figure 2.1.

In the preprocessing phase, a descriptor bit vector $B[c]$ encoding the transitions of the automaton is initialized for each character c of the alphabet. The bit in position i is set to zero in the bit vector if the i:th character in the pattern is c, in which case there is a transition on that character from state $i-1$ to state i in the automaton. Otherwise the bits are set to one.

The algorithm maintains a state vector E, which encodes the active states of the automaton with zeroes. In the beginning of the matching phase, the state vector E is initialized to 1^m. Then the text is read one character at a time from left to right, and the state vector is updated as follows:

$$E = (E \ll 1) \mid B[c] \ ,$$

where c is the character read. Shifting the bits left by one inserts a zero into the state vector which corresponds to the first state of the automaton always being active. Or'ing the bits with the preprocessed descriptor bit vector $B[c]$ corresponds to activating a state of the automaton if the previous state was active and the correct character was read from the text. If the m:th bit is zero after this update, the final state of the automaton is active, and thus we have found a match. The worst and average case complexity of shift-or is $\mathcal{O}(n)$ when the length of the pattern is less than or equal to the length of the computer word.

2.3.3 Backward Nondeterministic DAWG Matching

The Backward Nondeterministic DAWG Matching (BNDM) algorithm [79] has been developed from the Backward DAWG Matching (BDM) algorithm [32]. In the BDM algorithm, the pattern is preprocessed by forming a DAWG (directed acyclic word graph) of the reversed pattern. The text is processed in windows of size m, where m is the length of the pattern. The characters of the window are read from right to left, and using the DAWG, we search for the longest prefix of the pattern that matches a suffix of the window. When this search ends, we have either found a match (i.e. the longest prefix is of length m) or the longest prefix. If a match was not found, we can shift the start position of the window to the start position of the longest prefix. If a match was found, we can shift on the second longest prefix (the longest one is the match we just found).

The BNDM algorithm [79] is a bit-parallel simulation of the BDM algorithm. It uses a nondeterministic automaton instead of the deterministic one in the BDM algorithm. An example of such a nondeterministic automaton is shown in Figure 2.2. For each character c, a descriptor bit vector $B[c]$ encoding the transitions of the automaton is initialized in the preprocessing phase. The i:th bit is one in this vector if c appears in the reversed pattern in position i so that there is a transition from state $i-1$ to i on that character in the automaton. Otherwise the i:th bit is zero.

The algorithm maintains a state vector E, which encodes the active states of the automaton with ones. The state vector is initialized to $B[c]$, where c is the last character of the window. The same kind of right to left scan in a window of size m is performed as in the BDM algorithm. The state vector is updated in a similar fashion as in the shift-and algorithm [2, 11, 34]:

$$E = (E \ll 1) \ \& \ B[c] \ ,$$

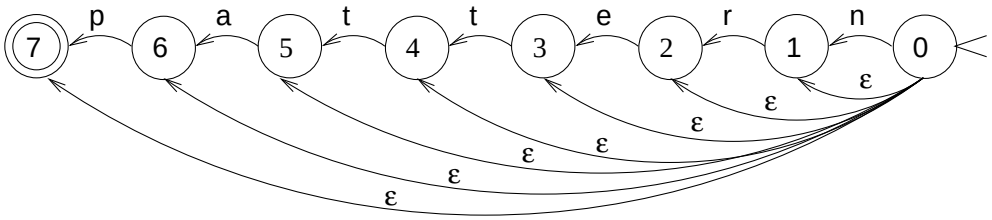


Figure 2.2: The nondeterministic automaton recognizing the reversed prefixes of the pattern "pattern"

where c is the character read. Shifting the bits to the left inserts a 0 to the vector, which corresponds to the first state being active only in the beginning, and and'ing the bits with the descriptor bit vector $B[c]$ corresponds to activating a state only if the previous state was active and the correct character was read. If the m:th bit is one after this update operation, we have found a prefix of length j, where j is the number of updates done in this window. If j is equal to m, a match has been found. If none of the states are active, i.e. $E = 0$, we can stop the scanning and shift the window according to the longest prefix we have found. The worst case complexity of the basic BNDM algorithm is $\mathcal{O}(nm)$. There are variations of BNDM with linear worst case complexity, but in practice, they are slower. The average case complexity of BNDM is $\mathcal{O}(n \log_\sigma m/m)$ when the length of the pattern is smaller than or equal to the length of the computer word.

The BNDM and shift-or algorithms use a very similar approach to encode the active states and to update the state vector. However, the BNDM algorithm encodes the active states with ones and uses an and operation to update the state vector, whereas the shift-or algorithm encodes the actives states with zeroes and uses an or operation to update the state vector. Encoding active states with zeroes is convenient in the shift-or algorithm because shifting the bits to the left by one introduces a zero to the least significant bit representing the first state of the automaton which is always active. In the BNDM algorithm, the first state is active only when we start handling a window, and thus it is convenient to represent active states with ones as then the first state automatically becomes nonactive after shifting the bits to the left.

2.3.4 Rabin-Karp Algorithm

The Rabin-Karp algorithm [54] uses a hash function from strings to integers to quickly discard most positions of the text. As preprocessing, the hash value of the pattern is computed. At each position i of the text, the hash value of the string $t_i \dots t_{i+m-1}$ is computed and compared to the hash value of the pattern. If these are equal, the position is verified by pairwise comparison. Karp and Rabin proposed to use a hash function $h(S)$ that can be quickly computed from the previous hash value $h(t_{i-1} \dots t_{i+m-2})$ and the next text character t_{i+m-1}. When using such a hash function, the time complexity

of the algorithm is $\mathcal{O}(n + occ \cdot m + focc \cdot m)$, where occ is the number of matches and $focc$ is the number of false matches, i.e. positions where the hash value matches the hash value of the pattern but the substring does not match the pattern.

2.4 Tools for Analysis

In this section, we will define the q-gram backward string matching algorithm and analyse its average case complexity. When analyzing the average case complexity, we assume that each character of the text and the pattern is chosen independently and uniformly at random.

The q-gram backward string matching algorithm does not solve a specific string matching problem nor does it present a complete algorithm, but rather the aim is to provide a framework of an algorithm that captures the behaviour of many algorithms for various string matching problems in such a way that we can analyse the asymptotic average case complexity. The framework captures the following important features of many string matching algorithms: The algorithm proceeds by aligning the pattern repeatedly with a *window* of text, and then shifting this window forward. The windows are read backward (i.e. from right to left), and the algorithm makes a maximal shift if the last q-gram of the window does not match the pattern in any position. An example of such an algorithm is the q-gram Boyer-Moore-Horspool algorithm adapted for various string matching problems.

The windows of text examined by a q-gram backward string matching algorithm are divided into good and bad windows. A window is good if the last q-gram of the window does not match the pattern in any position. All other windows are bad.

The *q-gram backward string matching algorithm* is defined by two constants, A and B, and three functions, $s(\cdot)$, $f(m, \cdot)$, and $g(q)$:

1. There is a constant $A > 0$ and a function $s(\cdot)$ such that the probability of a window to be bad is at most
$$\frac{m \cdot s(\cdot)}{\sigma^{Aq}} \ .$$
The constant A and the function $s(\cdot)$ depend on the string matching problem and also the arguments of the function $s(\cdot)$ depend on the string matching problem. In most cases, $A = 1$ and then $s(\cdot)$ is an upper bound on the number of q-grams that match a given q-gram.

2. The function $f(m, \cdot)$ gives the length of the shift after a good window. After a bad window, the algorithm shifts the window by at least one position. In most cases, $f(m, \cdot) = f(m, q) = m - q + 1$.

3. The function $g(q)$ gives the length of a q-gram. Thus, the last q-gram of a window is completely outside a previous window if the pattern has been shifted between the two windows by at least $g(q)$ positions. Clearly $g(q) \leq q$ as there

search ($T = t_1 \dots t_n$, n)

1. $i \leftarrow 1$
2. while ($i \leq n$)
3. algorithm specific processing
4. $i \leftarrow i + \text{shift}$

Figure 2.3: General q-gram backward string matching algorithm. Note that shift must be equal to $f(m, \cdot)$ if the window is good, i.e. the last q-gram of the window does not match the pattern in any position.

are q characters in a q-gram. For one-dimensional string matching algorithms, $g(q) = q$, but for higher dimensional algorithms, $g(q)$ is often less than q.

4. In a good window the work done by the algorithm is $\mathcal{O}(q)$.

5. There is a constant $B > 0$ such that the work done by the algorithm in a bad window is $\mathcal{O}(m^B \cdot s(\cdot)^B)$. The value of B often relates to the complexity of naive checking of a window.

The general pseudocode of the algorithm is given in Figure 2.3.

The following theorem is a useful tool for analyzing time complexities of q-gram backward string matching algorithms. The proof of the theorem is inspired by a similar proof for the Reverse Factor algorithm by Crochemore et al. [30].

Theorem 2.5. *The average case complexity of the q-gram backward string matching algorithm is*

$$\mathcal{O}\left(\frac{n}{f(m,\cdot)} \cdot q\right)$$

if $q > \frac{B+1}{A} \log_\sigma(m \cdot s(\cdot))$ *and* $g(q) \leq f(m, \cdot)$.

Proof. Let us divide the search phase of the algorithm into subphases. Let w_i, $i = 1, 2, \dots$, be the windows of the algorithm. The first subphase starts with w_1. Let w_s be the first window of a subphase. Then the first good window in the sequence $w_{s+g(q)\cdot k}$, $k = 0, 1, \dots$, is the last window of that subphase. If w_e is the last window of a subphase, then w_{e+1} starts a new subphase. Thus each subphase consists of X groups of $g(q)$ windows and one final good window, where $X \geq 0$ is a random variable. Each of the X groups of $g(q)$ windows starts with a bad window, and the rest $g(q) - 1$ windows in each of the X groups may be of any type. Figure 2.4 shows an example of dividing the windows into subphases.

The type of a window following a group of $g(q)$ windows is independent of the first window of the group, because the pattern has been shifted by at least $g(q)$ positions

$$\underbrace{\textbf{bad}\ \text{good}\ \textbf{good}}\ \underbrace{\textbf{good}}\ \underbrace{\textbf{bad}\ \text{bad}\ \textbf{bad}\ \text{good}\ \textbf{good}}\ \underbrace{\textbf{bad}\ \text{bad}\ \textbf{good}}\ \underbrace{\textbf{good}}\ \ldots$$

Figure 2.4: Dividing the search phase into subphases when $g(q) = 2$. The windows whose type influences the division are shown in boldface.

between them, and the type of a window is determined solely by the last q-gram of the window. If $f(m, \cdot) \geq g(q)$, the type of a window after a good window is also independent of the good window, i.e. the q-gram determining the type of the next window contains only characters that have not been previously read. Because each subphase contains at least one good window, the text of length n will be covered after $\mathcal{O}(n/f(m, \cdot))$ subphases.

We assumed that the probability of a bad window is at most $m \cdot s(\cdot)/\sigma^{Aq}$ and that the work done by the algorithm is $\mathcal{O}(q)$ in a good window and $\mathcal{O}(m^B \cdot s(\cdot)^B)$ in a bad window. Therefore, the expected work done by the algorithm in one subphase of searching will be less than

$$\begin{aligned}
& \mathcal{O}(q) \cdot P(X = 0) + \sum_{i=1}^{\infty} \left(\mathcal{O}(q) + i \cdot g(q) \cdot \mathcal{O}\left(m^B \cdot s(\cdot)^B\right)\right) \cdot P(X = i) \\
= \ & \mathcal{O}(q) + \sum_{i=1}^{\infty} i \cdot g(q) \cdot \mathcal{O}\left(m^B \cdot s(\cdot)^B\right) \cdot P(X = i) \\
\leq \ & \mathcal{O}(q) + q \cdot \mathcal{O}\left(m^B \cdot s(\cdot)^B\right) \sum_{i=1}^{\infty} i \cdot \left(\frac{m \cdot s(\cdot)}{\sigma^{Aq}}\right)^i \ .
\end{aligned}$$

This sum converges if $m \cdot s(\cdot)/\sigma^{Aq} < 1$ or equally if $q > (1/A) \log_\sigma(m \cdot s(\cdot))$, and then

$$\begin{aligned}
& \mathcal{O}(q) + q \cdot \mathcal{O}\left(m^B \cdot s(\cdot)^B\right) \sum_{i=1}^{\infty} i \cdot \left(\frac{m \cdot s(\cdot)}{\sigma^{Aq}}\right)^i \\
= \ & \mathcal{O}(q) + q \cdot \mathcal{O}\left(m^B \cdot s(\cdot)^B\right) \frac{\frac{m \cdot s(\cdot)}{\sigma^{Aq}}}{\left(1 - \frac{m \cdot s(\cdot)}{\sigma^{Aq}}\right)^2} \\
= \ & \mathcal{O}(q) + q \cdot \mathcal{O}\left(m^B \cdot s(\cdot)^B\right) \frac{m \cdot s(\cdot) \cdot \sigma^{Aq}}{(\sigma^{Aq} - m \cdot s(\cdot))^2} \ .
\end{aligned}$$

If we choose $q \geq C \log_\sigma(m \cdot s(\cdot))$, where $C > 1/A$ is a constant, then $\sigma^{Aq} \geq m^{AC} s(\cdot)^{AC}$. Because $AC > 1$, $\sigma^{Aq} - m \cdot s(\cdot) = \Omega(\sigma^{Aq})$, and therefore

$$\frac{1}{\sigma^{Aq} - m \cdot s(\cdot)} = \mathcal{O}\left(\frac{1}{\sigma^{Aq}}\right).$$

Now the work done by the algorithm in one subphase is less than

$$\begin{aligned} & \mathcal{O}(q) + q \cdot \mathcal{O}\left(m^B \cdot s(\cdot)^B\right) \frac{m \cdot s(\cdot) \cdot \sigma^{Aq}}{(\sigma^{Aq} - m \cdot s(\cdot))^2} \\ = \; & \mathcal{O}(q) \left(1 + \mathcal{O}\left(\frac{m^{B+1} s(\cdot)^{B+1} \sigma^{Aq}}{\sigma^{2Aq}}\right)\right) \\ = \; & \mathcal{O}\left(q \cdot \frac{m^{B+1} s(\cdot)^{B+1}}{\sigma^{Aq}}\right) \\ = \; & \mathcal{O}(q) \end{aligned}$$

if $C > (B+1)/A$.

There are $\mathcal{O}(n/f(m,\cdot))$ subphases and the average complexity of one subphase is $\mathcal{O}(q)$. Overall the average case complexity of the q-gram backward string matching algorithm is thus

$$\mathcal{O}\left(\frac{n}{f(m,\cdot)} \cdot q\right)$$

if $q > \frac{B+1}{A} \log_\sigma(m \cdot s(\cdot))$ and $g(q) \leq f(m,\cdot)$.

□

A corollary of the above theorem gives the average complexity of the q-gram Boyer-Moore-Horspool algorithm for exact string matching.

Corollary 2.6. *The average complexity of the q-gram Boyer-Moore-Horspool algorithm for exact string matching is $\mathcal{O}(n \log_\sigma m/m)$ for $q = \Theta(\log_\sigma m)$ and $m^2 < \sigma^m$.*

Proof. The q-gram Boyer-Moore-Horspool algorithm is a q-gram backward string matching algorithm with the following parameters. The probability of a bad window is equal to the probability that the last q-gram of the window matches the pattern in at least one position. This probability is at most m/σ^q as there are a total of σ^q different q-grams and less than m of them can occur in the pattern. Thus $A = 1$ and $s() = 1$. If the pattern has been shifted by at least q positions between two windows, then the last q-gram of the second window is completely outside the first one, and so $g(q) = q$. If a window is good, the algorithm clearly reads $\mathcal{O}(q)$ characters and makes a shift of length $f(m,q) = m - q + 1$. In a bad window, the algorithm will read the last q characters of the window to determine the shift length. Additionally, we need to determine if there is a match in that window. In the worst case, the last q characters of the window match because the window is bad, and the next q compared characters match because the previous shift was of length q. The average number of comparisons to determine if the rest of the pattern matches a random string is

$$\sum_{i=0}^{m-2q-1} \left(\frac{1}{\sigma}\right)^i = \frac{\sigma}{\sigma - 1}\left(1 - \frac{1}{\sigma^{m-2q}}\right)$$

because the $i+1$:st character of the random string is read only if all previous characters matched and the probability for this is $1/\sigma^i$ (see [10]). This is $\mathcal{O}(1)$ asymptotically for σ and m if $q \leq m/2$. In this case, the work done by the algorithm in a bad window is bounded by $2q + \mathcal{O}(1) = \mathcal{O}(q)$. If we choose $q = \mathcal{O}(\log_\sigma m)$, then $\mathcal{O}(q) = \mathcal{O}(\log_\sigma m) = \mathcal{O}(m^B)$ for any $B > 0$. By Theorem 2.5, if we choose $q > \frac{B+1}{A} \log_\sigma m = (B+1) \log_\sigma m$ such that $q \leq m-q+1$, then the average complexity of the q-gram Boyer-Moore-Horspool algorithm is $\mathcal{O}(nq/(m-q+1))$. The condition $q \leq m-q+1$ is equal to $q \leq (m+1)/2$, which always holds if the constraint $q \leq m/2$ holds. An appropriate q can be found if $\log_\sigma m < m/2$ or equally if $m^2 < \sigma^m$. If we choose $q = (B+1) \log_\sigma m + \epsilon = \Theta(\log_\sigma m)$, where $\epsilon > 0$ is a constant, the average case complexity becomes $\mathcal{O}(n \log_\sigma m/m)$. □

Yao [111] has proved that the lower bound for the average case complexity of the exact string matching problem for a single pattern is $\Omega(n \log_\sigma m/m)$, and so the q-gram Boyer-Moore-Horspool algorithm for exact string matching is average optimal for an appropriate choice of q.

Chapter 3

Approximate String Matching with Small Alphabets

In this chapter, we develop backward q-gram string matching algorithms for two variations of approximate string matching, the k-mismatch problem and the k-difference problem. Both of these problems are variations of the string matching problem, where the criteria for matching have been modified. In both of the problems, the pattern matches a substring of the text if the distance between the substring and the pattern is at most k. In the k-difference problem, the distance between two strings is the standard edit distance, where substitutions, deletions, and insertions are allowed. The k-mismatch problem is a more restricted one using the Hamming distance, where only substitutions are allowed. Chang and Marr [25] have proved that the lower bound for the average complexity of the approximate string matching problem is $\Omega(n(k + \log_\sigma m)/m)$, and they also give an algorithm that reaches this bound.

Several algorithms [75] for both variations of approximate string matching have been presented. For example, there are algorithms based on the dynamic programming table [24, 61, 95, 103, 104], bit parallel algorithms like the algorithm by Baeza-Yates and Navarro [13] and the Myers algorithm [73], and filtering algorithms including the approximate BNDM algorithm [79], the algorithm by Baeza-Yates and Perleberg [14], the algorithm by Sutinen and Tarhio [99], the approximate Boyer-Moore algorithm [101], and the algorithm by Fredriksson and Navarro [42]. Many of the algorithms have been developed with text data in mind, and these algorithms do not necessarily work well with a small alphabet. Our aim is to develop algorithms specifically for small alphabets, like DNA, which has lately attracted attention as approximate searching of large volumes of gene sequences has become common.

3.1 Preliminaries

3.1.1 Definitions

A *substitution* changes one character of a string into another character, an *insertion* inserts one character into any position of the string, and a *deletion* deletes one character from the string. The two most commonly used distance metrics in approximate string matching are the Hamming distance and the edit or Levenshtein distance [62].

Definition 3.1. *The* Hamming distance *of two strings of equal length, S and R, is the minimum number of substitutions needed to transform R into S.*

Definition 3.2. *The* edit distance *of two strings, S and R, is the minimum number of substitutions, insertions, and deletions needed to transform R into S.*

For example, the Hamming distance of the strings "cata" and "acta" is 2. Similarly, the edit distance of the strings "cata" and "cca" is 2.

The two approximate string matching problems studied in this chapter are then defined as follows.

Problem 3.3. *Given two strings, a text $T = t_1 \ldots t_n$ and a pattern $P = p_1 \ldots p_m$, and an integer k, the* k-mismatch problem *is to find all substrings of the text such that the Hamming distance between the pattern and the substring is at most k.*

Problem 3.4. *Given two strings, a text $T = t_1 \ldots t_n$ and a pattern $P = p_1 \ldots p_m$, and an integer k, the* k-difference problem *is to find all substrings of the text such that the edit distance between the pattern and the substring is at most k.*

Instead of reporting all the approximately matching substrings, most algorithms for the k-difference problem report either the starting or ending positions of occurrences. This is convenient because if the pattern matches a substring S with $i < k$ differences, then a substring S' starting at the same position but ending one position earlier or later matches with at most $i + 1 \leq k$ differences as we can transform S' into S with one deletion or insertion. Note, however, that the number of matches can vary depending on whether we report the starting or ending positions. Our algorithms report the ending positions of occurrences.

3.1.2 Dynamic Programming

Dynamic programming is a well known technique to calculate the edit distance between two strings, $R = r_1 \ldots r_m$ and $S = s_1 \ldots s_n$ [66, 82, 91, 94, 105, 106]. The dynamic programming table D of size $(m + 1) \times (n + 1)$ is initialized by setting $D[i, 0] = i$ for $0 \leq i \leq m$ and $D[0, j] = j$ for $0 \leq j \leq n$. The rest of the entries are filled with the recurrence relation:

$$D[i,j] = \min \begin{cases} D[i-1, j-1] + \alpha, \\ D[i-1, j] + 1, \\ D[i, j-1] + 1, \end{cases} \qquad \text{where } \alpha = \begin{cases} 0 & \text{if } r_i = s_j, \\ 1 & \text{otherwise.} \end{cases}$$

D

$i\backslash j$		t	g	g	c	a	a
	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
c 1	1	1	2	3	3	4	5
a 2	2	2	2	3	4	3	4
t 3	3	2	3	3	4	4	4
a 4	4	3	3	4	4	4	4

(a)

D

$i\backslash j$		t	g	g	c	a	a
	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
c 1	1	1	1	1	0	1	1
a 2	2	2	2	2	1	0	1
t 3	3	2	3	3	2	1	1
a 4	4	3	3	4	3	2	1

(b)

D

$i\backslash j$		t	g	g	c	a	a
	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
c 1	0	1	1	1	0	1	1
a 2	0	1	2	2	1	0	1
t 3	0	0	1	2	2	1	1
a 4	0	1	1	2	3	2	1

(c)

Figure 3.1: The edit distance table D for the strings R = "cata" and S = "tggcaa" with different initializations

Here the first alternative $D[i-1, j-1]+\alpha$ takes care of both substitutions and matches, and the second and third alternatives represent insertions and deletions. From this table, we can get the edit distances between the prefixes of R and the prefixes of S. In other words, the starting positions of the compared substrings are fixed to the beginning of the strings. The entry $D[m, n]$ gives the edit distance between the two strings. The entry $D[i, n]$, where $0 \leq i \leq m$, gives the edit distance between the prefix $r_1 \ldots r_i$ and the string S. Similarly, $D[m, j]$, where $0 \leq j \leq n$, gives the edit distance between the string R and the prefix $s_1 \ldots s_j$. Figure 3.1(a) gives an example of edit distance calculation for the strings "cata" and "tggcaa".

We can also initialize the first row to zero by setting $D[0, j] = 0$ for $0 \leq j \leq n$ and then fill the table with the same recurrence relation as before. Now deletions are free in the beginning of the string S, and thus $D[m, j]$ gives the minimum edit distance of aligning the string R against any substring of S ending at position j. Figure 3.1(b) shows an example of this kind of initialization. This initialization can be used to solve the k-difference problem by setting $R = P$ and $S = T$ [95]. An occurrence ending at position j is reported if $D[m, j] \leq k$.

It is also possible to initialize the table by setting $D[i, 0] = 0$ for $0 \leq i \leq m$ and $D[0, j] = 0$ for $0 \leq j \leq n$. Again we fill the table with the same recurrence relation as before. Now the starting position of either one, but not both, of the compared substrings can vary. $D[m, n]$ gives the minimum edit distance when aligning the string R against the string S, where deletions in the beginning of either R or S are free. Furthermore, the entry $D[m, j]$ gives the minimum edit distance when aligning the string R against the prefix $s_1 \ldots s_j$, where deletions in the beginning of either R or the prefix of S are free. Figure 3.1(c) shows an example of the edit distance table with this initialization.

Dynamic programming can also be used to calculate Hamming distances. Because the Hamming distance only allows substitutions, it is only possible to calculate the

		D						
		t	g	g	c	a	a	
$i\backslash j$	0	1	2	3	4	5	6	
	0	0	0	0	0	0	0	0
c	1	0	1	1	1	0	1	1
a	2	0	1	2	2	2	0	1
t	3	0	0	2	3	3	3	1
a	4	0	1	1	3	4	3	3

Figure 3.2: The Hamming distance table D for strings $R =$ "cata" and $S =$ "tggcaa".

Hamming distance of two strings of equal lengths. Thus only initializing the first row and column to 0 makes sense. Also the recurrence relation for filling the table is different:

$$D[i,j] = D[i-1,j-1] + \alpha, \quad \text{where } \alpha = \begin{cases} 0 & \text{if } r_i = s_j, \\ 1 & \text{otherwise.} \end{cases}$$

Suppose that $m \leq n$. Then the entry $D[m,n]$ gives the Hamming distance of aligning the string R against the end of S. The entries $D[m,j]$, where $0 \leq j \leq n$, give the Hamming distance for aligning the string R against $s_{j-m+1} \ldots s_j$. Figure 3.2 shows an example of the Hamming distance table for strings "cata" and "tggcaa".

3.1.3 Previous Algorithms

Here we will review algorithms based on the backward matching principle. For details on other algorithms, see the survey on approximate string matching by Navarro [75]. Many algorithms have been developed based on Boyer-Moore string matching [23] for the k-mismatch problem. Here we consider mainly ABM [101] and FAAST [65], but two other variations developed by Baeza-Yates and Gonnet [12], and El-Mabrouk and Crochemore [35] are worth mentioning.

The shift function of the Baeza-Yates-Gonnet algorithm generalizes the good suffix rule of the Boyer-Moore algorithm to the k-mismatch problem. We first observe that the number of mismatches between two strings is a metric distance obeying the triangular inequality. Thus, if the last j characters of the pattern match the text window with at most k mismatches and the last j characters of the pattern match another substring of the pattern with at least $2k+1$ mismatches, then the end of this text window will induce at least $k+1$ mismatches when aligned against this substring of the pattern. Therefore, we can shift the window so that this substring of the pattern is not aligned with the end of the previous window. As preprocessing we can then precompute the shifts by comparing the pattern against all possible shifts of the pattern and choosing for each j, $1 \leq j \leq m$, the minimum shift that will induce at most $2k$ mismatches between the j length suffix of the pattern and the shift of the pattern. The

Baeza-Yates-Gonnet algorithm achieves $\mathcal{O}(nk)$ average case complexity for searching and $\mathcal{O}(m(m-k))$ time for preprocessing.

The El-Mabrouk-Crochemore algorithm applies the Shift-Add approach [11]. However, the bit-parallel counters of shift-add are updated by reading characters in a window from right to left, and the processing of a window stops when all the counters have exceeded k. The algorithm then makes a shift based on the read characters, and the information already present in the counters is preserved. The average case complexity of the El-Mabrouk-Crochemore algorithm is $\mathcal{O}(n + kn/(m-k))$ for the searching phase, and its complexity for preprocessing is $\mathcal{O}(\sigma m^2)$.

The approximate Boyer-Moore (ABM) algorithm [101] is an adaptation of the Boyer-Moore-Horspool algorithm [49] to approximate matching. ABM uses the bad character heuristic for shifting and is thus a direct generalization of the Boyer-Moore-Horspool algorithm [49]. Instead of stopping at the first mismatch in the current window, the algorithm stops at the $k+1$:st mismatch or when an occurrence of the whole pattern is found. The shift is calculated considering last $k+1$ characters of the current window. The shift is the minimum of the precomputed shifts for those $k+1$ characters. After shifting, at least one of these characters will be aligned correctly with the pattern or the pattern will not be aligned with all these characters anymore. The average case complexity of searching in ABM is $\mathcal{O}(nk(1/(m-k) + k/\sigma))$, and the preprocessing cost is $\mathcal{O}(m + k\sigma)$.

ABM performs well on moderately large alphabets and low error levels although its average case time complexity is not optimal. Obviously, ABM was originally not designed for small alphabets, and in fact, it performs rather poorly on them. Liu et al. [65] tuned the k-mismatch version of ABM for smaller alphabets. Their algorithm, called FAAST, uses a stronger shift function based on a variation of the Four-Russians technique [8, 69, 109] to speed up the search. Instead of minimizing $k+1$ shifts during search, it generalizes the bad q-gram function for the k-mismatch problem and uses a precomputed shift table for the last q-gram of the window, where $q \geq k+1$ is a parameter of the algorithm. (The original paper used the notation $(k+x)$-gram.) The shift table is calculated so that after the shift at least $q-k$ characters are aligned correctly or the window is shifted past the last q-gram of the previous window. It is obvious that this stronger requirement leads to longer shifts in most situations when $q > k+1$, and the shift is never shorter than the shift of ABM. Note that for $q = k+1$ the length of the shift is the same for both the algorithms, but the shift is minimized during preprocessing in FAAST, while ABM performs the minimization of $k+1$ shifts during the search phase. So the algorithms are different even for $q = k+1$. The optimal value of q for maximum searching speed depends on other problem parameters and the computing platform. However, an increment of q makes the preprocessing time grow. FAAST presents a clear improvement on solving the k-mismatch problem for small alphabets as compared to the ABM algorithm. The preprocessing phase of FAAST is advanced because it includes the minimization step of ABM. The preprocessing cost of FAAST is $\mathcal{O}(q((m-k)\sigma^q + m))$, and the analysis in this work establishes the average

case complexity of searching in FAAST to be $\mathcal{O}(n(\log_\sigma m + k)/m)$, which has been shown to be optimal for approximate string matching [25].

A version of the ABM algorithm can also solve the k-difference problem, while the other Boyer-Moore type algorithms discussed above are limited to the k-mismatch problem. For the k-difference problem, the preprocessing cost of ABM is $\mathcal{O}((k+\sigma)m)$ and the average case complexity for searching in ABM is

$$\mathcal{O}\left(\frac{\sigma}{\sigma - 2k} kn \left(\frac{k}{\sigma + 2k^2} + \frac{1}{m}\right)\right)$$

if $2k + 1 < \sigma$.

Other approximate string matching algorithms utilizing the backward matching paradigm are approximate BNDM (ABNDM) [79] and the backward matching versions of the algorithm by Fredriksson and Navarro [42]. Both of these algorithms can solve both versions of the approximate string matching problem. ABNDM is a bit-parallel simulation of an automaton that identifies approximate matches of factors of the pattern in each alignment and then shifts the pattern according to the found matches. The k-difference version of ABNDM is a filtering algorithm, and so the found matches must be verified. The average case complexity of searching in ABNDM is

$$\mathcal{O}\left(n \frac{\alpha + \alpha^* \log_\sigma m/m}{(1-\alpha)\alpha^* - \alpha}\right) \ ,$$

where $\alpha = k/m$ is the error level and α^* is the maximum error level for which the probability of a random pattern matching a string with at most k differences is exponentially decreasing with m [75]. The preprocessing time of ABNDM is $\mathcal{O}(\sigma + m)$.

The algorithm by Fredriksson and Navarro [42] reads consecutive q-grams (ℓ-grams in the original paper) in a window and with the help of preprocessed tables determines the minimum number of mismatches or differences for aligning the q-grams with the pattern in some way. When the minimum number of mismatches or differences exceeds k, the window is shifted so that the first of these q-grams is not included in the new window. The potential matches must be verified. The algorithm has only been analyzed for the k-difference problem. The average case complexity of searching is $\mathcal{O}(n(k + \log_\sigma m)/m)$ if $k/m = 1/2 + \mathcal{O}(1/\sqrt{\sigma})$, and the complexity of preprocessing is $\mathcal{O}(m\sigma^q)$.

3.2 Algorithm for the k-Mismatch Problem

Our aim is to develop a faster algorithm for small alphabets based on FAAST, which uses a q-gram for shifting, where $q \geq k + 1$. We refine the usage of the Four-Russians technique [8, 69, 109] by making two major changes to FAAST. First, we implement a simpler and faster preprocessing phase based on dynamic programming. FAAST counts the number of mismatches in the last q-gram of the window during the searching

phase. Our second improvement is to compute this number during preprocessing, which improves the searching speed.

For each q-gram $G = g_1 \dots g_q \in \Sigma^q$, the preprocessing phase computes the Hamming distance when aligning the q-gram against the end of all prefixes of the pattern. As explained in Section 3.1.2, we get all these Hamming distances using dynamic programming by initializing the first row and column of the dynamic programming table to 0 and filling the rest of the table using the recurrence relation for Hamming distance. The bottom row $D[q, j]$, where $0 \leq j \leq m$, will then give the needed Hamming distances.

As an example, let us consider a situation where the pattern P = "tggcaa" has been aligned with the text window "gcata", and $k = 2$, $q = 4$ holds. The last q-gram of the window is now "cata", and the corresponding Hamming distance table of size $(q + 1) \times (m + 1)$, calculated during preprocessing, is shown in Figure 3.2. First of all, we see that the last cell $D[q, m] = 3 > k$, and therefore it is not possible to find a match at this position, as already the suffix of the window contains too many mismatches. Otherwise, we would have to check for a match by examining the amount of mismatches in the beginning of the window.

We will also look at the bottom row of the table and find the rightmost cell $D[q, j]$ with a value $h \leq k$, except for the last cell $D[q, m]$. This is the rightmost position of the pattern where the last q-gram of the current window matches the pattern with at most k mismatches, and thus the correct shift is equal to $m - j$. In our example, the rightmost cell with a value at most 2 is $D[q, 2] = 1$, and thus we would shift the window by $6 - 2 = 4$ positions.

As we do not need the whole table to obtain this information, we just store the calculated Hamming distance for each generated q-gram in a table M. The precalculated shifts are stored in a table S_q, which is a generalization of the bad q-gram function for approximate matching. During the searching phase, we read the last q-gram G of the window and check for an occurrence if $M[G] \leq k$. Finally, we shift the window according to $S_q[G]$.

We can improve the preprocessing time by applying the technique used previously by Fredriksson and Navarro [42] for approximate matching and Navarro et al. [81] for indexed approximate matching. If the q-grams are generated in the lexicographical order, the dynamic programming table differs only by the last few rows in most cases. Therefore, we can speed up the preprocessing if we only recalculate the last rows of the table at each step, starting from the first changed character.

This can be implemented by traversing the trie built of all q-grams in depth first order. Nodes at the i:th level of the trie correspond to strings of length i. Thus there are σ^i nodes on level i, and the total number of nodes in the trie is

$$\sum_{i=0}^{q} \sigma^i = \frac{\sigma^{q+1} - 1}{\sigma - 1} = \mathcal{O}(\sigma^q) \ .$$

If we have the dynamic programming table for a node in the trie, the tables for the children nodes can be obtained by calculating one more row to the dynamic program-

ming table, taking $\mathcal{O}(m)$ time per child. Thus calculating the dynamic programming tables for all nodes in the trie takes $\mathcal{O}(\sigma^q m)$ time. At the leaf nodes, we have the dynamic programming table for the corresponding q-gram, and we need to figure out the number of mismatches entered to table M and the shift value entered to table S_q, which takes $\mathcal{O}(m)$ time. The extra calculation needed at leaf nodes is thus $\mathcal{O}(\sigma^q m)$ because there are σ^q leaf nodes. Therefore, the time complexity of the preprocessing phase is $\mathcal{O}(\sigma^q m)$. Note that we do not need to explicitly build the trie if we implement the traversing of the trie by recursion. The preprocessing time of FAAST is $\mathcal{O}(q((m-k)\sigma^q + m))$, and therefore our preprocessing is asymptotically faster by a factor of q.

This algorithm for the k-mismatch problem is called FAAST2, and the pseudo code of the algorithm is shown in Figure 3.3. The shift behaviors of FAAST2 and FAAST are exactly the same. In FAAST, the number of mismatches in the last q-gram of an alignment is computed during the searching phase, whereas in FAAST2, this is fetched from a table. However, we still need to read the q-gram, and thus the time complexity of the search phase of FAAST2 is the same as in FAAST.

3.3 Algorithms for the k-Difference Problem

FAAST2 can be easily modified to solve the k-difference problem. As in FAAST2, we construct a dynamic programming table D' for each q-gram and the pattern during preprocessing. We do not store these tables but only use them to fill the tables M and S_q. To get the values for these tables, we need the edit distance of each q-gram aligned with the end of each prefix of the pattern. The example in Figure 3.4 shows the alignments of the pattern "tggcaa" with the q-gram "cata". As explained in Section 3.1.2, we get these edit distances by initializing the first row and column of the table D' to 0 and applying the recurrence relation for edit distance. The needed edit distances are then found in the last row $D'[q, j]$, where $0 \leq j \leq m$. For each q-gram, we store the minimum number of mismatches, insertions, and deletions needed to align the q-gram against the end of the pattern to the table M. This value is obtained from $D'[q, m]$. To enter the shift values to S_q, we find the largest $j < m$ such that $D'[q, j] \leq k$ and enter $m - j$ to the table S_q.

The searching phase now considers windows of length $m + k$ because the length of a match can vary as deletions and insertions are allowed. The searching phase starts by considering the text window ending at position $m-k$. In order to observe correctly an occurrence of the pattern in the beginning of the text, we assume that $t_{-2k+1}...t_0$ hold a character not in the pattern. When examining a window ending at position s, all matches ending before that position have been reported. If $M[t_{s-q+1} \dots t_s] \leq k$, we need to check for an occurrence ending at s by using dynamic programming. We initialize the table by setting $D[0, j] = 0$, where $0 \leq j \leq m + k$, and $D[i, 0] = i$, where $0 \leq i \leq m$, because we do not know the exact position of the start of the occurrence. As the maximum length of the occurrence is $m + k$, it is sufficient to

preprocess_helper ($P = p_1 \dots p_m$, m, k, q, i, G)

1. if ($i = q + 1$)
2. $\quad M[G] \leftarrow D[q, m]$
3. $\quad$ for ($j = m - 1$ down to 1)
4. $\quad\quad$ if ($D[q, j] \leq k$)
5. $\quad\quad\quad S_q[G] \leftarrow m - j$
6. $\quad\quad\quad$ break
7. else
8. $\quad$ for ($c \in \Sigma$)
9. $\quad\quad$ for ($j = 1$ to m)
10. $\quad\quad\quad D[i, j] \leftarrow D[i - 1, j - 1] + \alpha$, where $\alpha = \begin{cases} 0 \text{ if } c = p_j, \\ 1 \text{ otherwise} \end{cases}$
11. $\quad\quad$ preprocess_helper(P, m, k, q, $i + 1$, $G + c$)

preprocess ($P = p_1 \dots p_m$, m, k, q)

1. for ($i = 0$ to q)
2. $\quad D[i, 0] \leftarrow 0$
3. for ($j = 0$ to m)
4. $\quad D[0, j] \leftarrow 0$
5. preprocess_helper(P, m, k, q, 1, "")

search ($T = t_1 \dots t_n$, n, k, q)

1. $s \leftarrow m$
2. while ($s \leq n$)
3. $\quad$ if ($M[t_{s-q+1} \dots t_s] \leq k$) /* possible occurrence */
4. $\quad\quad c = M[t_{s-q+1} \dots t_s]$
5. $\quad\quad$ for ($i = 1$ to $m - q$)
6. $\quad\quad\quad$ if ($t_{s-q-i+1} \neq p_{m-q-i+1}$)
7. $\quad\quad\quad\quad c = c + 1$
8. $\quad\quad\quad\quad$ if ($c > k$) break
9. $\quad\quad$ if ($c \leq k$)
10. $\quad\quad\quad$ Report an occurrence at $t_{s-m+1} \dots t_s$ with c mismatches
11. $\quad s \leftarrow s + S_q[t_{s-q+1} \dots t_s]$

Figure 3.3: FAAST2 preprocessing and search phases

```
tggca-a      tggc--a      ..tggc
...cata      ...cata      cata--

..tgg        ..tg         ...t
cat-a        cata         cata
```

Figure 3.4: The alignments of the q-gram "cata" and the prefixes of the pattern "tggcaa" with minimum edit distance. The dots indicate free deletions and the hyphens normal deletions or insertions. These are examples of alignments with minimum edit distance. Also other alignments with the same edit distance are possible.

D'

		t	g	g	c	a	a
$i\backslash j$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0
c	1	0	1	1	1	0	1	1
a	2	0	1	2	2	1	0	1
t	3	0	0	1	2	2	1	1
a	4	0	1	1	2	3	2	1

(a)

D

		a	a	g	g	c	a	t	a
$i\backslash j$	0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0
t	1	1	1	1	1	1	1	0	1
g	2	2	2	2	1	1	2	2	1
g	3	3	3	3	2	1	2	3	2
c	4	4	4	4	3	2	1	2	3
a	5	5	4	4	4	3	2	1	2
a	6	6	5	4	5	4	3	2	2

(b)

D_{r}

		a	t	a	c	g	g	a	a
$i\backslash j$	0	1	2	3	4	5	6	7	8
	0	0	1	2	3	4	5	6	7	8
a	1	1	0	1	2	3	4	5	6	7
a	2	2	1	1	1	2	3	4	5	6
c	3	3	2	2	2	1	2	3	4	5
g	4	4	3	3	3	2	1	2	3	4
g	5	5	4	4	4	3	2	1	2	3
t	6	6	5	4	5	4	3	2	2	3

(c)

Figure 3.5: Normal and reversed edit distance tables for k-difference problem ($k = 2$, $q = 4$) with the pattern "tggcaa" and the test window "aaggcata". Sizes of the tables are $(q + 1) \times (m + 1)$ for D' and $(m + 1) \times (m + k + 1)$ for D and D_{r}.

construct a $(m + 1) \times (m + k + 1)$ edit distance table D with the current window $t_{s-(m+k)+1} \ldots t_s$ against the pattern. A match will be reported if $D[m, m + k] \leq k$. After this operation, we will shift the pattern according to S_q.

The preprocessing phase can be improved using the same technique as for FAAST2. The only difference is that we now use the recurrence relation for edit distance when filling the dynamic programming table. The modification of FAAST2 for the k-difference problem is called FAASTd.

Example tables for the k-difference problem are shown in Figures 3.5(a) and 3.5(b), using a pattern "tggcaa", a text window "aaggcata", and parameters $k = 2$ and $q = 4$. We can see from the first table that $S_q[\text{"cata"}] = 6-5 = 1$ and $M[\text{"cata"}] = D'[q, m] = 1$. Therefore, we would construct a table D, find that $D[m, m+k] = 2 \leq k$, and report a match. We would then continue the search by shifting the window by one position.

In the k-mismatch problem, we did not need to reread the last q characters from the window when checking for an occurrence. Instead, we had stored the number of

mismatches in the table M, and we could extend the match based on that information. For the k-difference problem, the situation is not quite as simple because we need to compute the dynamic programming table to check for an occurrence. The problem with FAASTd is that the window is read forward when checking for an occurrence, while during the preprocessing phase, we have generated the dynamic programming table for the last characters of the pattern. In order to use that information and avoid rereading the last q characters, we need to reverse the calculation of the dynamic programming table so that we start building the table from the end of the pattern and the text window.

Suppose that we want to check for an occurrence ending at position s. We can build the edit distance table for the reversed pattern $p_m \dots p_1$ and the reversed text substring $t_s \dots t_{s-(m+k)+1}$. For the reversed strings, the starting position of the occurrence is fixed, so we initialize the table by setting $D_{\mathrm{r}}[0, j] = j$ and $D_{\mathrm{r}}[i, 0] = i$ for $i \in [0, m], j \in [0, m + k]$. This reversed table gives equivalent results when it comes to calculating the actual edit distance between the pattern and the window. When this reversed edit distance table D_{r} has been finished, we have to search for a match at the last row. To be exact, we need to check $2k + 1$ different cells of the table for a possible match because the match can contain up to k insert or delete operations, and the match length can therefore vary. All possible matches that end in the character t_s will be found in the last $2k + 1$ cells of the last row of the reversed table. We can either report the first match with at most k differences or search for the match with the minimum differences. The current window $t_{s-(m+i)+1} \dots t_s$ matches the pattern $p_1 \dots p_m$ with at most k differences if

$$D_{\mathrm{r}}[m, m + i] \leq k$$

for any $i \in -k \dots k$. Figure 3.5(c) shows an example of the reversed edit distance table.

To avoid rereading the last q-gram of a window for constructing the edit distance table, we can calculate the reversed edit distance table for each q-gram and the pattern during preprocessing. During searching, we can then check for a complete occurrence by filling the rest of the table columns from t_{s-q} down to $t_{s-(m+k)+1}$. We can therefore store the last column of the reversed table $D_{\mathrm{r}}[j, q]$, $j \in [0, m]$, for each q-gram during the preprocessing phase. This column can then be used to fill up the rest of the table by dynamic programming during the search phase when the window needs to be checked for an occurrence, and thus we do not need to run dynamic programming for the whole table every time. However, we still need the normal edit distance table to obtain the values for tables S_q and M.

When verifying an occurrence, we need the last $2k+1$ columns to be able to check the cells $D_{\mathrm{r}}[m, m + i]$, where $i \in -k \dots k$. Thus only the first of these columns can be computed during preprocessing, and so we must choose $q \leq m - k$.

We modify FAASTd to use the reversed table during the search phase, and we also store the last column of the reversed tables generated during the preprocessing phase. The new algorithm is called FAASTd2, and its pseudo code is given in Figure 3.6.

For simplicity, the preprocessing part of the pseudo code does not use the optimization of generating the q-grams in lexicographic order and recalculating the dynamic programming table only for those rows that have changed.

The preprocessing phase of FAASTd has the same time complexity as that of FAAST2, as the only difference is that the dynamic programming table is filled using the recurrence relation for edit distance in FAASTd, while the preprocessing phase of FAAST2 uses the recurrence relation for Hamming distance. In FAASTd2, we need to calculate both the original dynamic programming table and the reversed one. Because a q-gram is read in opposite directions when calculating these two tables, we have to enumerate the q-grams twice. However, the asymptotic time complexity remains the same.

FAASTd and FAASTd2 degenerate to calculating the dynamic programming table at each position of the text in the worst case. Thus the worst case complexity of FAASTd and FAASTd2 is $\mathcal{O}(nm^2)$.

The worst case complexity can be improved to $\mathcal{O}(nm)$ by doing the verification forward and incrementally. This technique has previously been used to improve the worst case complexity of ABM [101] as well as many other approximate string matching algorithms based on filtering [75]. We now store the end position of the previous verification and also the last column of the dynamic programming table of the previous verification. When we need to perform another verification, we first check if the starting position of this new verification is before the end position of the previous verification. If so, we continue the verification from the end position of the previous verification. This guarantees that we never traverse a text position twice for verification purposes. Now in the worst case, the pattern is always shifted by one position and a verification is triggered in each position. To compute the shifts, we read $\mathcal{O}(qn)$ characters, and the verification cost in each position is $\mathcal{O}(m)$, as we add one new column to the dynamic programming table. Thus the worst case complexity is $\mathcal{O}(nm)$. This modification of the algorithm is called FAASTdw.

To compute a new column to the dynamic programming table, we only need the previous column of the table, so it is enough to save the previous and current columns of the table. The space needed for verification in FAASTdw is thus $\mathcal{O}(m)$.

3.4 Analysis

In the worst case, FAAST and FAAST2 will read $\mathcal{O}(m)$ characters in each window, and thus their worst case complexity is $\mathcal{O}(nm)$. The worst case complexity of FAASTd and FAASTd2 is $\mathcal{O}(nm^2)$, and the worst case complexity of FAASTdw is $\mathcal{O}(nm)$ as stated above.

Let us then analyse the average case complexity of the algorithms. Here we assume the standard random string model, where each character of the text and the pattern is chosen independently and uniformly at random. We will make use of the following

preprocess ($P = p_1 \dots p_m$, m, k, q)

1. for ($i = 0$ to q)
2. $\quad D'[i, 0] \leftarrow 0$
3. $\quad D_{\mathrm{r}}[0, i] \leftarrow i$
4. for ($j = 0$ to m)
5. $\quad D'[0, j] \leftarrow 0$
6. $\quad D_{\mathrm{r}}[j, 0] \leftarrow j$
7. for ($G = g_1 \dots g_q \in \Sigma^q$)
8. $\quad$ for ($i = 1$ to q)
9. $\quad\quad$ for($j = 1$ to m)
10. $\quad\quad\quad D'[i, j] \leftarrow \min \begin{cases} D'[i-1, j-1] + \alpha, \\ D'[i-1, j] + 1, \\ D'[i, j-1] + 1, \end{cases} \quad \alpha = \begin{cases} 0 \text{ if } g_i = p_j, \\ 1 \text{ otherwise} \end{cases}$
11. $\quad\quad\quad D_{\mathrm{r}}[j, i] \leftarrow \min \begin{cases} D_{\mathrm{r}}[j-1, i-1] + \alpha, \\ D_{\mathrm{r}}[j-1, i] + 1, \\ D_{\mathrm{r}}[j, i-1] + 1, \end{cases} \quad \alpha = \begin{cases} 0 \text{ if } g_{q-i+1} = p_{m-j+1}, \\ 1 \text{ otherwise} \end{cases}$
12. $\quad M[G] \leftarrow D'[q, m]$
13. $\quad \mathrm{lastColumn}[G] \leftarrow D_{\mathrm{r}}[0 \dots m, q]$
14. $\quad$ for ($j = m - 1$ down to 1)
15. $\quad\quad$ if ($D'[q, j] \leq k$)
16. $\quad\quad\quad S_q[G] \leftarrow m - j$
17. $\quad\quad\quad$ break

search ($T = t_1 \dots t_n$, n, k, q)

1. for ($i = q + 1$ to $m + k$)
2. $\quad D_{\mathrm{r}}[0, i] \leftarrow i$
3. $s \leftarrow m - k$
4. while ($s \leq n$)
5. $\quad$ if ($M[t_{s-q+1} \dots t_s] \leq k$) /* possible occurrence */
6. $\quad\quad D_{\mathrm{r}}[0 \dots m, q] \leftarrow \mathrm{lastColumn}[t_{s-q+1} \dots t_s]$
7. $\quad\quad$ for($i = q + 1$ to $m + k$)
8. $\quad\quad\quad$ for ($j = 1$ to m)
9. $\quad\quad\quad\quad D_{\mathrm{r}}[j, i] \leftarrow \min \begin{cases} D_{\mathrm{r}}[j-1, i-1] + \alpha, \\ D_{\mathrm{r}}[j-1, i] + 1, \\ D_{\mathrm{r}}[j, i-1] + 1, \end{cases} \quad \alpha = \begin{cases} 0 \text{ if } t_{s-i+1} = p_{m-j+1}, \\ 1 \text{ otherwise} \end{cases}$
10. $\quad\quad$ if ($D_{\mathrm{r}}[m, m + i] \leq k, i \in -k \dots k$)
11. $\quad\quad\quad$ Report match at $t_{s-(m+i)+1} \dots t_s$ with $D_{\mathrm{r}}[m, m + i]$ differences
12. $\quad s \leftarrow s + S_q[t_{s-q+1} \dots t_s]$

Figure 3.6: FAASTd2 preprocessing and search phases

Lemmas originally proved by Chang and Marr [25], but here we use the rewritten form by Fredriksson and Navarro [42], which is more convenient for our purposes.

Lemma 3.5. *The probability that two random q-grams have a common subsequence of length $(1-c)q$ is at most $a\sigma^{-dq}/q$ for constants $a = (1+o(1))/(2\pi c(1-c))$ and $d = 1-c+2c\log_\sigma c + 2(1-c)\log_\sigma(1-c)$. The probability decreases exponentially for $d > 0$, which surely holds if $c < 1 - e/\sqrt{\sigma}$.*

Lemma 3.6. *If G is a q-gram that matches inside a given string P (longer than q) with less than cq differences, then G has a common subsequence of length $q - cq$ with some q-gram of P.*

Theorem 3.7. *If $k/m < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$, the average case complexity of FAAST, FAAST2, FAASTd, FAASTd2, and FAASTdw is $\mathcal{O}(n(\log_\sigma m + k)/m)$ if we choose $q = \Theta(\log_\sigma m + k)$.*

Proof. Let us choose a constant c such that $d > 0$, and therefore the probability of matching given by Lemma 3.5 decreases exponentially when q increases. Given the number of mismatches or differences $k \leq cq$, the probability that a q-gram matches inside the pattern with at most k mismatches or differences is at most $ma/(q\sigma^{dq})$, because there are less than m q-grams in the pattern, and by Lemma 3.6 one of them has to have a common subsequence of length $q - k \geq q - cq$ with the q-gram, and the probability for this event is given by Lemma 3.5.

We will now show that FAAST, FAAST2, FAASTd, FAASTd2, and FAASTdw are q-gram backward string matching algorithms as defined in Section 2.4. The length of a q-gram is clearly q, and so $g(q) = q$. A window is bad if the last q-gram of the window matches the pattern in any position with at most k mismatches (for FAAST and FAAST2) or differences (for FAASTd, FAASTd2, and FAASTdw). This probability is at most $ma/(q\sigma^{dq}) < ma/\sigma^{dq}$, so $s() = a$ and $A = d$.

In a good window, all the algorithms will read the last q characters of the window and conclude that a shift of length $f(m,q) = m - q + 1$ can be made. Thus the work done in a good window is bounded by $\mathcal{O}(q)$. If the window is bad and the last q-gram of the window matches the end of the pattern with at most k mismatches, FAAST and FAAST2 will compare the rest of the pattern against the text and count the number of mismatches. Thus the work in bad windows can be bounded by $\mathcal{O}(m)$ in FAAST and FAAST2, and therefore $B = 1$. In FAASTd, FAASTd2, and FAASTdw, the dynamic programming table will be built if the last q-gram of the window matches the end of the pattern with less than k differences, and so the work on bad windows is bounded by $\mathcal{O}(m^2)$, and so $B = 2$.

By Theorem 2.5, the average case complexity of FAAST, FAAST2, FAASTd, FAASTd2, and FAASTdw is therefore $\mathcal{O}(nq/(m-q+1))$ if we choose $q > (B+1)/d\log_\sigma(ma)$ such that $q \leq (m-q+1)$, which is equal to $q \leq (m+1)/2$. Thus $m - q + 1 = \Omega(m)$, and the average case complexity is then $\mathcal{O}(nq/m)$. Additionally, $q \geq k/c$, so a safe choice for q is $q = (B+1)/d\log_\sigma(ma) + k/c = \Theta(\log_\sigma m + k)$, and

then the average case complexity of the algorithms is $\mathcal{O}(n(\log_\sigma m+k)/m)$. An appropriate q exists if $k/c < (m+1)/2$ and $(B+1)/d \log_\sigma(ma) < (m+1)/2$. The condition $k/c < (m+1)/2$ becomes $k/(m+1) < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$ if we choose $c < 1 - e/\sqrt{\sigma}$, which also assures that $d > 0$. The latter condition $(B+1)/d \log_\sigma(ma) < (m+1)/2$ is equal to $\sigma^{m+1} > (ma)^{2(B+1)/d}$, which holds asymptotically for σ and m. □

The lower bound for the average complexity of the approximate string matching problem was proved to be $\Omega(n(\log_\sigma m + k)/m)$ by Chang and Marr [25], and thus FAAST and the new algorithms presented here are average optimal for an appropriate choice of q when $k/m < 1/2 - \mathcal{O}(1/\sqrt{\sigma})$.

Recall that the preprocessing time of our algorithms is $\mathcal{O}(m\sigma^q)$. With the choice $q = \Theta(k + \log_\sigma m)$, the preprocessing time becomes $\mathcal{O}(\sigma^{\Theta(k)} m^{\Theta(1)})$. The preprocessing time of FAAST is $\mathcal{O}(q((m-k)\sigma^q + m))$, which becomes $\mathcal{O}((\log_\sigma m + k)((m-k)\sigma^{\mathcal{O}(k)} m^{\mathcal{O}(m)} + m))$ with the choice $q = \Theta(\log_\sigma m + k)$.

FAAST2 uses two preprocessed tables, M and S_q, both of which are of size σ^q. During the preprocessing phase, we will additionally need space for the dynamic programming table, which requires $\mathcal{O}(mq)$ space. Thus the total space requirement for FAAST2 is $\mathcal{O}(\sigma^q + mq) = \mathcal{O}(\sigma^{\Theta(k)} m^{\Theta(1)} + m \log_\sigma m + mk)$.

FAASTd uses similar preprocessed tables and a similar dynamic programming table in the preprocessing phase as FAAST2, but additionally it needs to build the dynamic programming table in the searching phase requiring an extra $\mathcal{O}(m^2)$ space. Thus, the space complexity of FAASTd is $\mathcal{O}(\sigma^{\Theta(k)} m^{\Theta(1)} + m \log_\sigma m + mk + m^2)$.

FAASTd2 adds another structure to those of FAASTd. It also stores the last row of the dynamic programming table for each q-gram. The additional space needed for this is $\mathcal{O}(m\sigma^q) = \mathcal{O}(\sigma^{\Theta(k)} m^{\Theta(1)})$. Thus the asymptotic space complexity of FAASTd2 is also $\mathcal{O}(\sigma^{\Theta(k)} m^{\Theta(1)} + m \log_\sigma m + mk + m^2)$.

FAASTdw uses the same structures as FAASTd except that it stores only the two last columns of the dynamic programming table during searching. Thus the asymptotic space complexity is $\mathcal{O}(\sigma^{\Theta(k)} m^{\Theta(1)} + m \log_\sigma m + mk)$.

3.5 Experimental Results

Tests were run on an AMD Athlon 1.0 GHz dual core CPU with 2 GB of memory, 64 kB L1 cache, and 512 kB L2 cache. The computer was running Linux 2.6.23. The algorithms were written in C and compiled with the gcc compiler. For comparison in the k-mismatch case, we used the following algorithms:

- ABM: The original ABM algorithm.
- FAAST: Our implementation of FAAST.
- FN: The mismatch version of the algorithm by Fredriksson and Navarro [42].

For the k-difference problem, we compared FAASTd and FAASTd2 against the following algorithms:

- ABM: A version of ABM for the k-difference problem.
- Myers: Myers algorithm [73] is a linear time bit-parallel algorithm for patterns shorter than the computer word.
- BYP: The algorithm by Baeza-Yates and Perleberg [14] divides the pattern into smaller pieces. If the pattern now occurs at some position, at least one of the pieces must have an exact occurrence at that position. The algorithm then searches for exact matches of the pieces and verifies the occurrences found by the exact search. We use the implementation by Baeza-Yates and Navarro [13], which adapts the Sunday algorithm [98] for searching the pattern pieces.
- FN: The algorithm by Fredriksson and Navarro [42] for the k-difference problem.

All the results are shown with the q-value gaining the fastest searching speed in FAAST and our new algorithms if not otherwise stated. The best q-value is generally the same for our algorithms and for FAAST. We tried several versions of the single pattern algorithm by Fredriksson and Navarro generally getting the best results with the version that reads the window backwards (`-Sb` option). Also for this algorithm we show the results with the best value for the parameter q. The other algorithms do not utilize the parameter q.

The searched text is a 22 MB sequence of the fruit fly genome. The patterns have been extracted randomly from the text. Each pattern set consists of 200 different patterns of the same length, and they are searched sequentially.

Table 3.1 shows the search times for the original ABM, FAAST, FN, and FAAST2 in the k-mismatch problem, and Figure 3.7 further illustrates the results. We used the code by Fredriksson and Navarro to measure the times for the FN algorithm. The code is designed for multiple patterns, and the precision of measuring preprocessing time is not good enough to get reliable results for a single pattern. Thus the preprocessing times for the FN algorithm are not shown. As can be seen, FAAST2 is the fastest for this setting, and it is generally about 30% faster than FAAST in the k-mismatch case for $k \in [1, 2]$. Also, the preprocessing phase of FAAST2 is 10 to 30 times faster than that of FAAST.

Experimental results for the k-difference problem are shown in Table 3.2, and Figure 3.8 further illustrates the results. In the k-difference problem, our new algorithms are faster than the Myers, BYP, and ABM algorithms. They are also faster than the FN algorithm with short patterns, but with longer patterns, the FN algorithm is faster. The basic version of the Myers algorithm is limited by the 32-bit word size, and it cannot handle patterns with $m > 32$. The modifications in FAASTd2 decrease search time by 20-30% when compared to FAASTd.

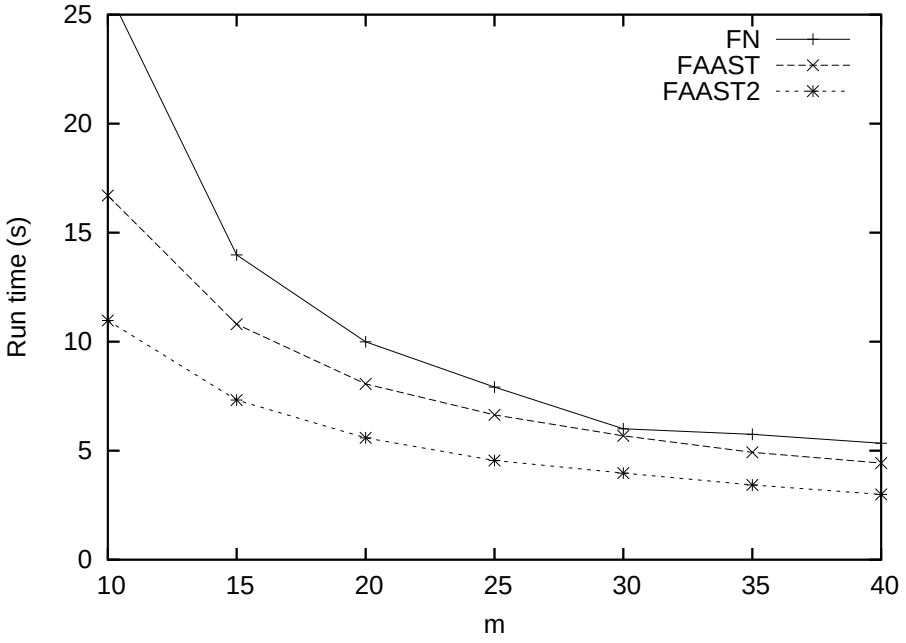
Table 3.1: Search times in seconds for k-mismatch, using best observed q-values. The corresponding preprocessing times are shown in parenthesis. The runtimes of the algorithm by Fredriksson and Navarro shown here are for the options `-Sb`. The algorithms was slightly faster with options `-Sb -O` for $m = 10$, $k = 2$ yielding the runtime 73.83.

(a) $k = 1$

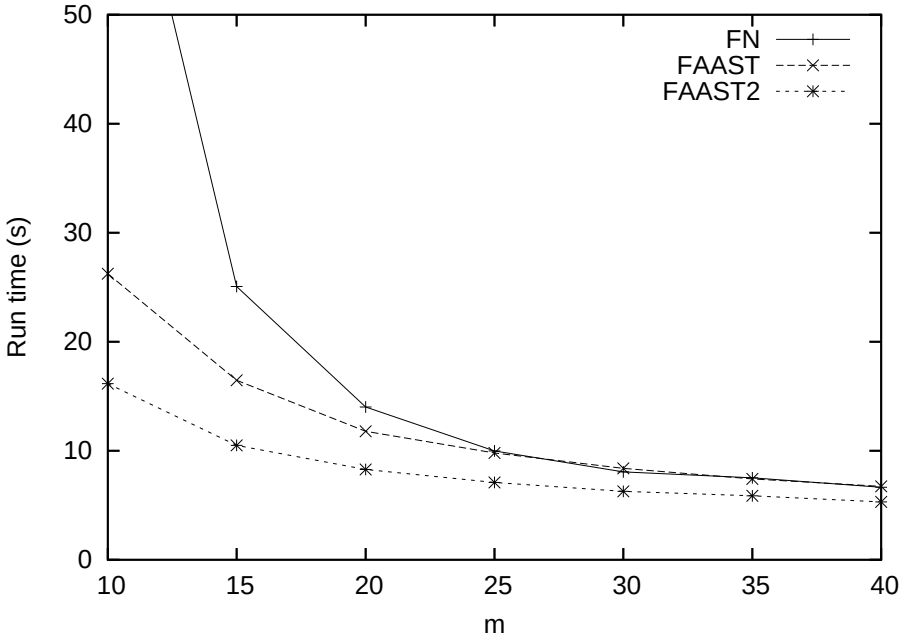
	ABM	FAAST		FN		FAAST2	
m	runtime (s)	runtime (s)	q	runtime (s)	q	runtime (s)	q
10	53.98 (0.03)	16.70 (0.15)	5	26.11	5	10.97 (0.02)	5
15	51.16 (0.01)	10.80 (1.39)	6	13.98	5	7.32 (0.02)	5
20	50.57 (0.06)	8.06 (2.01)	6	9.99	6	5.59 (0.14)	6
25	51.57 (0.04)	6.64 (2.26)	6	7.92	6	4.55 (0.16)	6
30	51.69 (0.10)	5.68 (3.27)	6	6.00	6	3.97 (0.12)	6
35	51.59 (0.17)	4.92 (4.03)	6	5.75	6	3.43 (0.20)	6
40	49.51 (0.21)	4.43 (4.75)	6	5.34	5	2.99 (0.35)	6

(b) $k = 2$

	ABM	FAAST		FN		FAAST2	
m	runtime (s)	runtime (s)	q	runtime (s)	q	runtime (s)	q
10	102.72 (0.02)	26.25 (0.62)	6	74.63	5	16.15 (0.13)	6
15	99.92 (0.02)	16.46 (5.88)	7	25.06	7	10.50 (0.17)	6
20	95.30 (0.05)	11.79 (8.16)	7	14.01	9	8.28 (0.14)	6
25	97.23 (0.06)	9.80 (10.74)	7	9.99	8	7.09 (0.18)	6
30	96.88 (0.13)	8.38 (13.41)	7	8.06	8	6.28 (0.22)	6
35	95.59 (0.09)	7.41 (16.00)	7	7.51	9	5.86 (0.16)	6
40	94.49 (0.24)	6.74 (18.66)	7	6.65	9	5.30 (1.21)	7



(a) $k = 1$



(b) $k = 2$

Figure 3.7: Runtime comparison of the algorithms for the mismatch problem

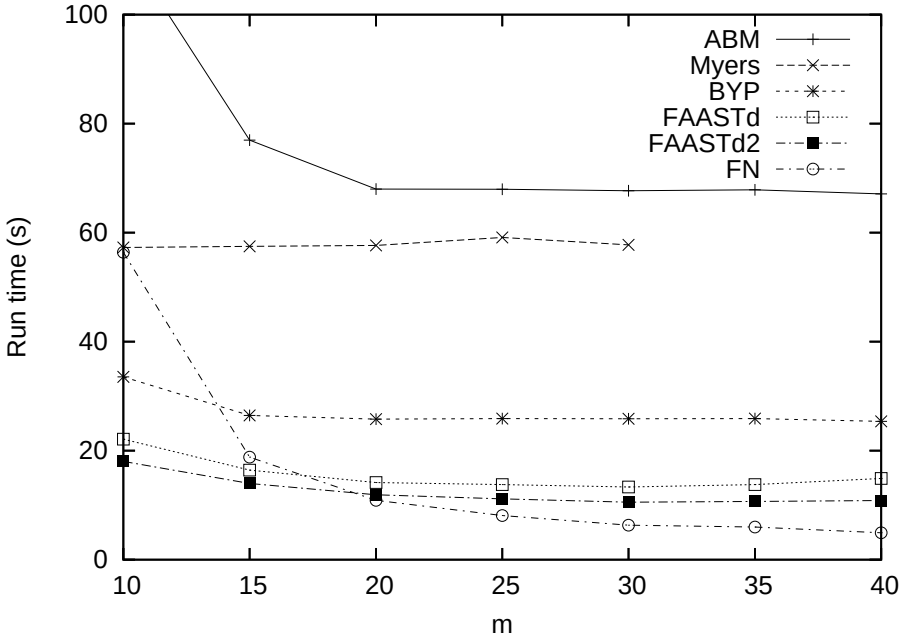
Table 3.2: Search times in seconds for k-difference, using best observed q-values. The runtimes of the algorithm by Fredriksson and Navarro shown here are for the options `-Sb`. For $k = 1$, the algorithm was slightly faster with options `-Sb -L` for $m = 10$ yielding the runtime 52.07. For $k = 2$, the algorithm was faster with options `-Sf` for $m = 10$ and with options `-Sb -O` for $m = 15$ yielding the runtimes 112.30 and 59.98, respectively.

(a) $k = 1$

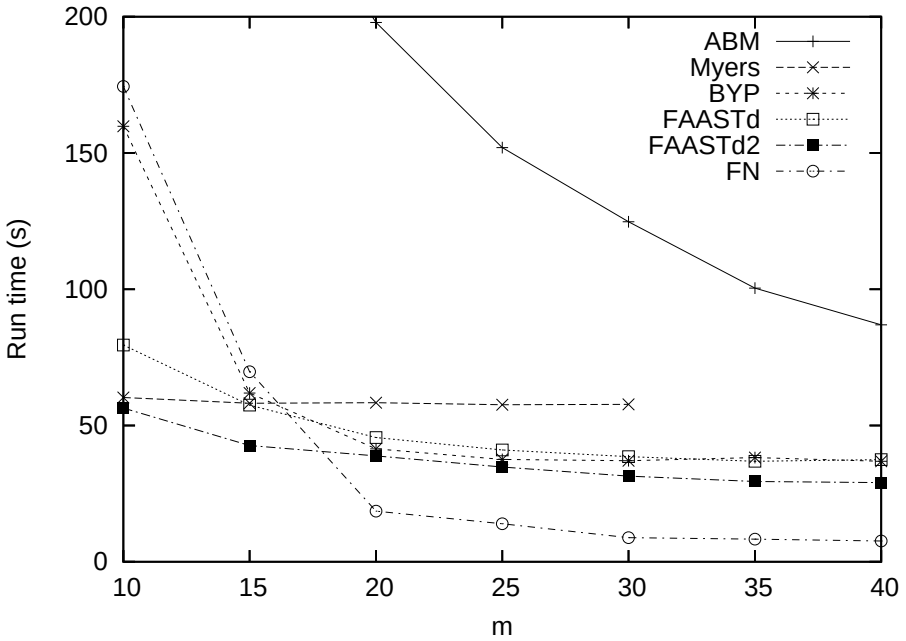
	ABM	Myers	BYP	FN		FAASTd		FAASTd2	
m	runtime (s)	runtime (s)	runtime (s)	runtime (s)	q	runtime (s)	q	runtime (s)	q
10	112.78	57.29	33.53	56.41	4	22.11	7	18.04	6
15	76.97	57.47	26.45	18.81	6	16.43	7	13.98	7
20	68.00	57.67	25.79	10.89	6	14.15	7	11.92	7
25	67.97	59.10	25.89	8.10	6	13.79	7	11.17	7
30	67.70	57.76	25.86	6.34	6	13.35	7	10.56	7
35	67.85	-	25.89	5.99	6	13.79	7	10.67	7
40	67.12	-	25.37	4.92	6	14.91	7	10.84	7

(b) $k = 2$

	ABM	Myers	BYP	FN		FAASTd		FAASTd2	
m	runtime (s)	runtime (s)	runtime (s)	runtime (s)	q	runtime (s)	q	runtime (s)	q
10	392.37	60.30	159.78	174.42	8	79.51	8	56.41	8
15	268.95	58.15	61.93	69.75	6	57.50	9	42.65	8
20	197.90	58.38	41.43	18.58	9	45.55	9	38.85	9
25	151.94	57.65	37.54	13.92	9	41.02	9	43.76	9
30	124.78	57.76	37.10	8.84	9	38.56	9	31.42	9
35	100.43	-	38.23	8.27	9	36.89	9	29.47	9
40	86.95	-	36.85	7.62	9	37.48	9	29.07	9



(a) $k = 1$



(b) $k = 2$

Figure 3.8: Runtime comparison of the algorithms for the difference problem

Table 3.3: Preprocessing times and search times for k-difference with different q-values ($k = 1, m = 20$)

	Preprocessing		Search	
q	FAASTd	FAASTd2	FAASTd	FAASTd2
2	<0.01	0.01	4578.31	3500.39
3	0.01	0.02	1053.72	748.52
4	0.03	0.05	236.29	158.90
5	0.04	0.15	66.19	44.76
6	0.23	0.57	23.72	16.59
7	1.27	2.37	14.15	11.92
8	4.83	9.78	19.08	17.14
9	19.29	39.80	28.23	28.13
10	77.06	159.41	34.51	34.54

The effect of increasing the precalculated edit distance table size, and thus increasing preprocessing time with a large q-value is shown in Table 3.3. With small values of q, the search time decreases as the amount of preprocessing increases, but after a certain limit, increasing the q-value will begin to slow down the search. For these pattern lengths and k-values, the optimal q-value was typically $k + 4$ for the k-mismatch problem and $k + 6$ for the k-difference problem.

In the implementation of FAASTd, preprocessing is optimized by generating the q-grams in lexicographic order and recalculating the dynamic programming table only for those characters that differ from the previous q-gram, while FAASTd2 needs to do this recursion twice, once to generate the normal dynamic programming table and once to calculate the reversed one. Thus the preprocessing times in Table 3.3 are longer for FAASTd2 than for FAASTd.

Chapter 4

Parameterized String Matching

In this chapter, we explore string matching under another matching criterion, parameterized matching [16]. Two strings are a parameterized match if the characters of the first string can be renamed in such a way that it is transformed into the second string. The parameterized string matching problem is a variation of the string matching problem, where all substrings of the text that are a parameterized match with the pattern should be reported.

The parameterized string matching problem has been investigated in two dimensions by Amir et al. [5] and Hazay et al. [48]. Other related work includes parameterized matching of multiple patterns [51], parameterized matching with mismatches [47], and approximate parameterized search [18].

Previous research of parameterized string matching has been focused on developing algorithms with good worst-case performance. Some effort to develop an algorithm fast on average was made by Baker [17], who developed an algorithm based on the Boyer-Moore algorithm [23], but the average case complexity was not analyzed. Fredriksson and Mozgovoy [41] have also recently developed sublinear algorithms for one dimensional parameterized string matching.

In this chapter, we introduce q-gram backward string matching algorithms for both the one-dimensional and two-dimensional parameterized string matching problems. We analyze the time complexities of the algorithms for random texts and moderately repetitive patterns. The experimental results confirm the results of the analysis and show our algorithms to be fast in practice.

4.1 Definitions

Definition 4.1. *Two strings, $S = s_1 \dots s_m$ and $R = r_1 \dots r_m$, drawn from an alphabet Σ are a* parameterized match *(or* p-match *for short) if there exists a bijection $\pi : \Sigma \mapsto \Sigma$ such that for each i, $s_i = \pi(r_i)$.*

Strings "abac" and "bcba" are a p-match because the bijection $\pi(a) = c, \pi(b) = a, \pi(c) = b$ transforms "bcba" into "abac". On the other hand, strings "aabb" and

"acbb" are not a p-match because a bijection cannot map both 'a' and 'c' to 'a', and thus there is no bijection that can transform "acbb" to "aabb".

Problem 4.2. *Given a text $T = t_1 \dots t_n$ and a pattern $P = p_1 \dots p_m$ in an alphabet Σ, the* parameterized string matching problem *is to find all substrings of the text that are a p-match with the pattern.*

Problem 4.3. *Given a text T of size $n \times n$ and a pattern P of size $m \times m$, the* two-dimensional parameterized string matching problem *is to find all those $m \times m$ substrings of the text that are a p-match with the pattern.*

Two disjoint alphabets were used in the original definition of the parameterized string matching problem by Baker [16]. One of the alphabets was a fixed alphabet like in the standard string matching problem, and the other one was a parameterized alphabet like our Σ. Both the pattern and the text could contain characters from both alphabets, but characters from the fixed alphabet were required to match exactly. We decided to use only the parameterized alphabet because that is natural for the two dimensional problem of image search, and we wished to give a unified treatment to both the one dimensional and two dimensional cases.

Many of the algorithms make use of so called *predecessor strings*. A string S is transformed into a predecessor string as follows. If a character in position i has occurred previously in the string in position j and j is the most recent such position, the position i in the predecessor string contains $i - j$. Otherwise the predecessor string contains 0. For example, the string "aabac" is transformed into 0-1-0-2-0. Now it can be fairly easily seen that two strings are a p-match if and only if their predecessor strings match exactly [16].

Another way to transform the two strings so that the transformed strings will match exactly if the original strings were a p-match is to replace all occurrences of the first occurring character with 1, the second one with 2, and so on. For example, the string "aabac" is transformed into 1-1-2-1-3. The resulting sequence of integers is called a restricted growth function.

Definition 4.4. *A* restricted growth function *(RGF) of length m is a sequence of m integers, $s_1, \dots, s_m$, satisfying the following criteria:*

$$\begin{aligned} s_1 &= 1 \\ s_i &\leq \max\{s_1, \dots, s_{i-1}\} + 1, \qquad \textit{if } 2 \leq i \leq m \ . \end{aligned}$$

The properties of restricted growth functions have been studied previously, see e.g. Kreher and Stinson [59]. There are b_k different RGFs of length k, where b_k is the k:th Bell number, which is defined as follows:

$$b_k = \sum_{i=1}^{k} \frac{1}{i!} \sum_{j=1}^{i} (-1)^{i-j} \binom{i}{j} j^k \ .$$

RGFs can also be ranked. A ranking algorithm for RGFs determines the position of a given RGF with regard to some order. In our case, the exact ordering imposed by the ranking algorithm is not relevant. We just need to get a unique integer for each RGF. When ranking RGFs, we have used the ranking algorithm described in Kreher and Stinson [59], which runs in $\mathcal{O}(q)$ time, where q is the length of the RGF.

Definition 4.5. *A pattern is* (q, ℓ)-repetitive *if in all* q*-grams of the pattern, at least* ℓ *characters have occurred previously in that* q*-gram, i.e. there are at most* $q - \ell$ *distinct characters in the* q*-gram.*

The pattern "aaaa" is $(2, 1)$-repetitive, while the pattern "aabb" is $(3, 1)$-repetitive but not $(2, 1)$-repetitive because the substring "ab" contains no repetition. Similarly, a two-dimensional pattern is (q^2, ℓ)-repetitive if for all substrings of size $q \times q$ (a two-dimensional q-gram), at least ℓ of the characters have occurred earlier in that substring.

4.2 Earlier Solutions

4.2.1 One-Dimensional Algorithms

In her original paper, Baker [16] gave a suffix tree based algorithm for finding parameterized matches. The algorithm first preprocesses both the text and the pattern by transforming them into predecessor strings. After this preprocessing, the problem can almost be solved by conventional exact string matching algorithms. The only remaining problem is that if we are considering a window on the text, the predecessor pointers might point to positions outside the window. Baker proposed modifications to the suffix tree construction algorithm that take care of this problem. The resulting construction algorithm runs in $\mathcal{O}(n \log n)$ time. The construction of the suffix tree was further improved by Kosaraju [58], who developed an algorithm with time complexity $\mathcal{O}(n(\log \lambda + \log \sigma))$, where σ and λ are the sizes of the parameterized and the fixed alphabet. Cole and Hariharan [28] also further explored the construction of the suffix tree and developed a randomized linear time algorithm.

Baker [17] has also proposed a Boyer-Moore based algorithm, which uses predecessor strings. The algorithm is a modification of the TurboBM algorithm [30] using predecessor strings to find p-matches. The worst case time complexity of the algorithm is $\mathcal{O}(n \log \min(m, \sigma))$. The average case complexity of the algorithm was not studied in the paper.

Amir et al. [6] have proposed an algorithm for the parameterized string matching problem based on the Knuth-Morris-Pratt algorithm [57] for exact string matching. Their algorithm runs in the worst case in $\mathcal{O}(n \log \sigma)$ time. They also prove that their algorithm is optimal in the worst case if the alphabet is unbounded.

Fredriksson and Mozgovoy [41] have also developed sublinear algorithms for one-dimensional parameterized matching. Their algorithms are based on the shift-or [11] and backward DAWG matching (BDM) [30] algorithms. The shift-or based algorithm

runs in $\mathcal{O}(n\lceil m/w \rceil)$ worst case time with average case complexity $\mathcal{O}(n \log_\lambda m/w)$, where w is the size of the computer word and λ is the size of the fixed alphabet, and the BDM based algorithm has average case complexity $\mathcal{O}(n \log_\lambda m/m)$. The BDM based algorithm can also be modified to search for multiple patterns simultaneously. The average case analysis of these algorithms relies on the text containing a substantial fraction of symbols from the non-parameterized alphabet.

4.2.2 Two-Dimensional Algorithms

The two-dimensional parameterized matching problem was first considered by Amir et al. [5] in the context of function matching. They give an algorithm that preprocesses the text into a predecessor representation suitable for two-dimensional strings and then applies a conventional two-dimensional algorithm. The worst case running time of the algorithm is $\mathcal{O}(n^2 \log^2 m)$. Hazay et al. [48] give another algorithm for two-dimensional parameterized matching that is based on the "duel-and-sweep" paradigm. In the worst case, this algorithm runs in $\mathcal{O}(n^2 + m^{2.5}\text{polylog}(m))$ time. Both of these algorithms are quite complicated, and neither one of them has been implemented as far as we know.

4.3 Horspool Style Algorithms

In this section, we describe q-gram backward string matching algorithms for parameterized matching. Our algorithms are generalizations of the q-gram Boyer-Moore-Horspool algorithm.

4.3.1 Three One-Dimensional Algorithms

We need to make two changes to the Boyer-Moore-Horspool algorithm to adapt the q-gram Boyer-Moore-Horspool algorithm for parameterized matching. First of all, we need to modify the checking of the window to recognize p-matches instead of exact matches. Secondly, the algorithm for parameterized matching must shift the window so that the last q-gram of the window is a p-match with the pattern after the shift.

The recognition of parameterized matches when checking a window can be done in a straightforward way. During preprocessing, we transform the reversed pattern into a predecessor string, and when checking for a match during searching, we transform the reversed window of text into a predecessor string and compare these two predecessor strings.

To shift the window correctly in the parameterized algorithm, we need to redefine the bad q-gram function $S_q[G]$. The bad q-gram function for parameterized matching is defined as follows:

$$S_q[G] = \min\{h \mid p_{m-h-q+1} \dots p_{m-h} =_\text{p} G, 1 \leq h \leq m-q\} \ ,$$

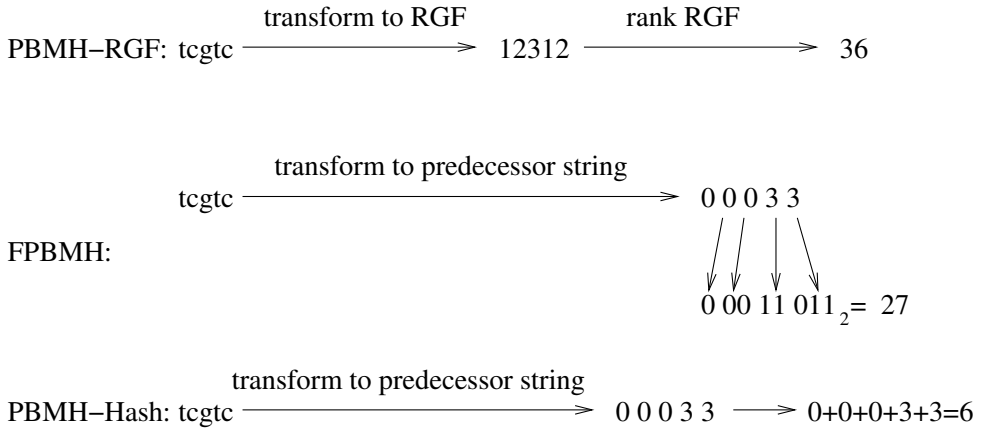


Figure 4.1: Transforming the 5-gram "tcgtc" into an index in PBMH-RGF, FPBMH, and PBMH-Hash algorithms.

where $=_{\mathrm{p}}$ denotes parameterized matching. If the q-gram is not a p-match with any q-gram of the prefix of the pattern $p_1 \dots p_{m-1}$, $S_q[G] = m - q + 1$. Now we could enumerate all q-grams and for each q-gram find the rightmost (except for the last) q-gram of the pattern that is a p-match with it. We would then store this information for all possible q-grams. However, it turns out that we need to use larger q-grams with the parameterized matching algorithm than with the exact one, and thus it is not practical to store the shifting information for all q-grams. To solve this problem, we note that all q-grams that are a p-match with each other give the same shift. Thus, we can use the predecessor strings or RGFs to index the shift table. An obvious solution is to use the rank of the RGFs as indexes. We call this algorithm Parameterized Boyer-Moore-Horspool with RGF or PBMH-RGF for short.

Figure 4.1 gives an example of the index calculation. First, the q-gram "tcgtc" is transformed to a restricted growth function by replacing all 't':s, which is the first occurring character, with 1:s, all 'c':s with 2:s, and all 'g':s with 3:s. Then we apply the RGF ranking algorithm to the resulting RGF "12312", getting a result of 36. This number can then be used to index the shift table.

The problem with this approach is that although calculating the rank of an RGF of length q can be done in $\mathcal{O}(q)$ time, there is a fairly large constant in the time complexity, and this operation needs to be done for each inspected window. Another alternative for calculating the indexes is to transform the q-gram into a predecessor string and then to reserve enough bits for each character of the predecessor string in the index. The i:th character of the predecessor string takes values between 0 and $i-1$, and so $\lceil \log_2 i \rceil$ bits are needed to represent it. The index thus has a constant width of $s = \sum_{i=2}^{q} \lceil \log_2 i \rceil$.

Figure 4.1 shows an example of the index calculation for this case too. First, we transform the q-gram "tcgtc" to a predecessor string. As the first character 't' has not appeared before, it is transformed to 0. Similarly, the second character 'c' and the third character 'g' are transformed to 0:s. The fourth character is a 't', which has occurred

Table 4.1: The number of entries in the shift table for PBMH-RGF, FPBMH, and PBMH-Hash for various values of q.

Algorithm	2	3	4	5	6	7	8	9	10
PBMH-RGF	2	5	15	52	203	877	4 140	21 147	115 975
FPBMH	2	8	32	256	2 048	16 384	131 072	2 097 152	33 554 432
PBMH-Hash	2	4	7	11	16	22	29	37	46

previously in position one. Since that was three positions ago, the fourth element in the predecessor is 3. Similarly, the last character 'c' is replaced with a 3. Now the resulting predecessor string "00033" is transformed into an index. No bits are reserved for the first character because it is always the same, and thus it is not used in the calculation. One bit is reserved for the second character, and so the first bit in the index will be 0. The third character uses two bits, and thus we transform the 0 into bits 00, and so on. The resulting index is 00011011_2 in binary, which represents the number 27.

We call this algorithm Fast Parameterized Boyer-Moore-Horspool or FPBMH for short. This approach wastes space, but the indexes are much faster to calculate. The RGF approach needs a table of size b_q, where b_q is the q:th Bell number, while the FPBMH algorithm needs a table of size 2^s, where $s = \sum_{i=2}^{q} \lceil \log_2 i \rceil$. Table 4.1 shows the number of entries in the shift table for both approaches for different values of q.

In a random text, the distribution of the predecessor strings is very steep. The most common predecessor string of length q, 0^q, has a high probability if the alphabet is reasonably large, while the least common predecessor string, 01^{q-1}, has a probability close to 0. Therefore we might need to use quite large q-grams, which is a problem for FPBMH. On the other hand, hashing the q-grams cleverly might let us use even larger q-grams than the PBMH-RGF algorithm can handle. For those q-grams that have the same hash value, the minimum shift will be stored in the shift table, and so the shifts will be somewhat shorter than without hashing. We tried hashing the q-grams by transforming them first to predecessor strings and then adding up all the positions of the predecessor string.

In the example of Figure 4.1, the q-gram "tcgtc" is transformed into an index using this hashing scheme. First, the q-gram is transformed into the predecessor string "00033" exactly like in the FPBMH algorithm. Next, we add up all the characters of the predecessor string, yielding the index value 6.

With this hashing scheme, the most common q-gram is the only one hashed to 0, and thus the hashing might even out the distribution of the q-grams. The value of the hash function is surely at most

$$0 + 1 + \ldots + (q-1) = \sum_{i=1}^{q-1} i = \frac{q(q-1)}{2} \ ,$$

and thus the table size is $q(q-1)/2+1$. This modification of the algorithm is called PBMH-Hash. Table 4.1 also includes the space requirement for this approach.

4.3.2 A Two-Dimensional Algorithm

The two-dimensional algorithm is based on the two-dimensional string matching algorithm by Tarhio [100], which is a cross of the Boyer-Moore-Horspool algorithm and the Kärkkäinen-Ukkonen algorithm [53]. In the algorithm by Tarhio, the text is divided into $\lceil (n-m)/m \rceil + 1$ strips, each of which has m columns. Each strip is then searched with a Boyer-Moore-Horspool type algorithm, and each potential match is verified with the trivial algorithm.

In each position, the character at the lower right hand corner is investigated. If this character occurs in the lowest row of the pattern, there is a potential match, which has to be verified. These are found with the help of two tables, M and N. $M[c]$ is the column where the character c occurs first in the lowest row of the pattern, and N links the occurrences of c in the lowest row of the pattern. The pattern is shifted down the strip with another table S, which is a generalization of the bad character function:

$$S[c] = \min\{h \mid \exists i \text{ s.t. } p_{m-h,i} = c, 1 \leq h \leq m-1, 1 \leq i \leq m\} \ .$$

If c does not appear in the first $m-1$ rows of the pattern, $S[c] = m$.

The algorithm can be modified to read several characters and calculate the shifts based on all these characters. If we read $q \times q$ characters (a two-dimensional q-gram), the text will be divided into $\lceil (n-m)/(m-q+1) \rceil + 1$ strips, each containing $m-q+1$ columns.

This algorithm, which uses q-grams, can fairly easily be extended to parameterized matching in a similar fashion as the Boyer-Moore-Horspool algorithm was extended for one-dimensional parameterized matching. The resulting algorithm proceeds exactly like the algorithm by Tarhio, but the read q-grams are transformed into predecessor strings, and these are then used to index the tables. To transform the two-dimensional q-gram into a predecessor string, we first transform it into a one-dimensional string by concatenating the rows. This string can then be transformed to a predecessor string, which is further used to index the tables. As with the one-dimensional case, there are several ways to transform the predecessor strings into indexes. We implemented the transformation the same way as in the FPBMH algorithm.

4.4 Analysis

We first analyze the worst and average case complexity of the one-dimensional algorithms and then turn to the two-dimensional case. When analyzing the average case complexity, we assume the standard random string model, where each character of the text is chosen independently and uniformly at random.

4.4.1 The One-Dimensional Algorithms

The preprocessing phase of the algorithms consists of initializing the shift table, which takes time proportional to the number of entries in the table. Additionally, to preprocess the pattern, we need to keep track of where the different symbols of the alphabet occurred previously, and thus the preprocessing of the q-grams of the pattern takes $\mathcal{O}(\sigma + mq)$ time, where σ is the size of the alphabet. As stated earlier, the number of entries in the shift table is b_q for PBMH-RGF, 2^s for FPBMH, and $q(q-1)/2+1$ for PBMH-Hash, where b_q is the q:th Bell number and $s = \sum_{i=2}^{q} \lceil \log_2 i \rceil$. The following lemma gives a nice formulation to the space complexity of FPBMH:

Lemma 4.6. *If $q \geq 2$, then $2^s \leq q^{q-1}$, where $s = \sum_{i=2}^{q} \lceil \log_2 i \rceil$.*

Proof. When $q = 2$, it holds that $2^{\sum_{i=2}^{q} \lceil \log_2 i \rceil} = 2 = 2^{2-1}$, and so the Lemma holds when $q = 2$.

Let us then assume that the Lemma holds for the value q. Now with the value $q+1$, we get

$$2^{\sum_{i=2}^{q+1} \lceil \log_2 i \rceil} = 2^{\sum_{i=2}^{q} \lceil \log_2 i \rceil} \cdot 2^{\lceil \log_2 (q+1) \rceil} \leq q^{q-1} \cdot 2^{1+\log_2 q} = 2 \cdot q^q \ .$$

Here we have used the assumption that the Lemma holds for the value q, and thus $2^{\sum_{i=2}^{q} \lceil \log_2 i \rceil} \leq q^{q-1}$, and the inequality $\lceil \log_2 (q+1) \rceil \leq 1 + \log_2 q$.

We know that the function $\left(\frac{q+1}{q}\right)^q = (1+1/q)^q$ is an increasing function, which approaches Napier's constant as q approaches infinity. When $q = 2$, $(1+1/q)^q = 2.25$, and thus $2 \leq \left(\frac{q+1}{q}\right)^q$, when $q \geq 2$. Therefore,

$$2^{\sum_{i=2}^{q+1} \lceil \log_2 i \rceil} \leq 2 \cdot q^q \leq \left(\frac{q+1}{q}\right)^q \cdot q^q = (q+1)^q \ .$$

This proves that if the Lemma holds for the value q, it also holds for the value $q+1$. Since the Lemma also holds for $q = 2$, by induction the Lemma holds for all $q \geq 2$. □

Therefore, the preprocessing phases of PBMH-RGF, FPBMH, and PBMH-Hash have time complexities $\mathcal{O}(b_q + \sigma + mq)$, $\mathcal{O}(q^{q-1} + \sigma + mq)$, and $\mathcal{O}(q^2 + \sigma + mq)$, respectively.

The only difference in the matching phase of our algorithms is how q-grams are used to index the shift table. In both PBMH-RGF and FPBMH, two q-grams are transformed into the same index if and only if they are a p-match, and the transformation is done in $\mathcal{O}(q)$ time. Therefore, the matching phases of PBMH-RGF and FPBMH algorithms have the same time complexities. The hashing in the PBMH-Hash algorithm slightly changes the time complexity of the algorithm since two q-grams are sometimes transformed into the same index even if they are not a p-match. However, such collisions are sufficiently rare with large alphabets, and so the analysis holds also for PBMH-Hash when the alphabet is large.

In the worst case, the one-dimensional algorithms find a match in each window, and the length of the shift is always one, yielding a total of $n - m + 1$ windows. In each window, all characters are read and compared to the pattern. Thus, the worst case complexity of PBMH-RGF, FPBMH, and PBMH-Hash is $\mathcal{O}(nm)$.

Let us then analyze the average case complexity. In order to do that, we need to consider the probability distribution of the different predecessor strings corresponding to random q-grams. Let σ denote the size of the alphabet, and let z be the number of zeroes in the given predecessor string. Because the predecessor string of a q-gram is also of length q, clearly $z \leq q$. Each of the zeroes presents a different character in the original string, and each non-zero element of the predecessor string is defined by the zeroes. Because each zero represents a different character and there are σ characters in the alphabet, it must also hold that $z \leq \sigma$. The characters corresponding to the zeroes in the predecessor string can be chosen in $\sigma \cdot (\sigma - 1) \cdot \ldots \cdot (\sigma - z + 1)$ ways, and there are a total of σ^q different strings. Thus, the probability that the given predecessor string of length q and with z zeroes matches the predecessor string of a random string is

$$\frac{\sigma \cdot (\sigma - 1) \cdot \ldots \cdot (\sigma - z + 1)}{\sigma^q} = \frac{\sigma!}{\sigma^q \cdot (\sigma - z)!} \ .$$

Theorem 4.7. *If we choose a $q \leq m/2$ such that the pattern is $(q, \log_\sigma m)$-repetitive, then the average case complexity of searching in PBMH-RGF and FPBMH is $\mathcal{O}(nq/m)$.*

Proof. PBMH-RGF and FPBMH are q-gram backward string matching algorithms as defined in Section 2.4 with the following parameters. The length of a q-gram is clearly q, and thus $g(q) = q$. The probability that a random q-gram is a p-match with the pattern in any position is less than

$$\frac{m \cdot \sigma!}{\sigma^q \cdot (\sigma - z)!} < \frac{m}{\sigma^{q-z}} \ ,$$

where z is the maximum number of zeroes in any q-gram of the pattern. This is also the probability that a window is bad, and thus $s(\sigma, z) = \sigma^z$ and $A = 1$. Because we have chosen q so that the pattern is $(q, \log_\sigma m)$-repetitive, $z \leq q - \log_\sigma m$. Clearly the algorithms will make a shift of length $f(m, q) = m - q + 1$ after a good window, and the work done by the algorithms in a good window is $\mathcal{O}(q)$, because in a good window the last q-gram of the window does not match the pattern in any position. If the window is bad, then in the worst case the last q characters of the window match, and the previous q characters match because of the previous shift. Because $q > z$, at least every q:th element of the predecessor string of the pattern is not zero. The probability of matching for these non-zero elements is $1/\sigma$. Thus, the average number of characters read by the algorithms in a bad window is at most

$$2q + \sum_{i=0}^{\lfloor m/q \rfloor - 3} q \cdot \frac{1}{\sigma^i} = 2q + q \cdot \frac{\sigma}{\sigma - 1} \left(1 - \frac{1}{\sigma^{\lfloor m/q \rfloor - 2}} \right),$$

which is asymptotically $\mathcal{O}(q)$ if $q \leq m/2$. Because the pattern is $(q, \log_\sigma m)$-repetitive, $q = \mathcal{O}(z + \log_\sigma m)$, and then the work done by the algorithms in a bad window is $\mathcal{O}(z + \log_\sigma m) = \mathcal{O}(\sigma^{Bz} m^B) = \mathcal{O}(s(\sigma, z)^B m^B)$ for any $B > 0$. By Theorem 2.5, the average complexity of PBMH-RGF and FPBMH is then $\mathcal{O}(nq/(m - q + 1)) = \mathcal{O}(nq/m)$ if $q > (1+B)(\log_\sigma m + z)$ for any constant $B > 0$ such that $q \leq m - q + 1$. The condition $q \leq m - q + 1$ is equal to $q \leq (m + 1)/2$, which always holds if the constraint $q \leq m/2$ holds. An appropriate q can be found if there is a $q \leq m/2$ such that the pattern is $(q, \log_\sigma m)$-repetitive. □

If we have both a fixed and a parameterized alphabet, the preprocessing time of the algorithms will change slightly because the size of the shift table will depend on the fixed alphabet also. The preprocessing phase of PBMH-RGF, FPBMH, and PBMH-Hash will have time complexities $\mathcal{O}(\lambda^q b_q + \sigma + mq)$, $\mathcal{O}(\lambda^q q^{q-1} + \sigma + mq)$, and $\mathcal{O}(\lambda^q q^2 + \sigma + mq)$, respectively, where λ is the size of the fixed alphabet. The above analysis for the average time complexity holds also in this case. In fact, the fixed alphabet makes the problem easier. In this case, the average case complexity of the algorithms is $\mathcal{O}(nq/m)$ if we choose a $q \leq m/2$ such that at least $\max(\log_\sigma m, \log_\lambda m)$ characters in each q-gram of the pattern are either from the fixed alphabet or have occurred earlier in the q-gram.

4.4.2 The Two-Dimensional Algorithm

Let us first consider the complexity of the preprocessing phase. The two-dimensional algorithm uses the strategy of the FPBMH algorithm when calculating the indexes of the shift table. Thus, the number of entries in the shift table is 2^s, where $s = \sum_{i=2}^{q^2} \lceil \log_2 i \rceil$. As with the one-dimensional algorithms, we also need to keep track of the previous occurrences of the alphabet symbols, and thus a table of size σ is needed for that. Therefore, the complexity of the preprocessing phase of the two-dimensional algorithm is $\mathcal{O}((q^2)^{q^2-1} + \sigma + m^2 q^2)$.

The worst case for the two-dimensional algorithm occurs when all the $(n-m+1)^2$ windows of the text match the pattern, and thus the worst case time complexity of the two dimensional algorithm is $\mathcal{O}(n^2 m^2)$.

Theorem 4.8. *If we choose a $q \leq (m + 1)/2$ such that the pattern is $(q^2, \log_\sigma m^2)$-repetitive, then the average case complexity of searching in the two-dimensional algorithm is $\mathcal{O}(n^2 q^2/m^2)$.*

Proof. Let us consider the time complexity of the algorithm when it matches the pattern against one strip of the text. Then the two-dimensional algorithm is a q^2-gram backward string matching algorithm as defined in Section 2.4 with the following parameters. The length of a q^2-gram is q, and so $g(q^2) = q < q^2$. The probability that a random q-gram is a p-match with the pattern in any position is less than

$$\frac{m^2 \cdot \sigma!}{\sigma^{q^2} \cdot (\sigma - z)!} < \frac{m^2}{\sigma^{q^2 - z}} \ ,$$

where z is the maximum number of zeroes in the predecessor string of any q-gram in the pattern. This is also the probability of a bad window, and thus $s(m, \sigma, z) = m\sigma^z$ and $A = 1$.

Clearly the work done by the algorithm in a good window is $\mathcal{O}(q^2)$, and the algorithm makes a shift of length $f(m, q) = m - q + 1$ after a good window. If the window is bad, then on average only in $\mathcal{O}(1)$ alignments the last q-gram matches because the pattern is $(q^2, \log_\sigma m)$-repetitive. Furthermore, in the worst case also the previous q-gram in that alignment matches because of the previous shift. Thus, the complexity of a bad window is

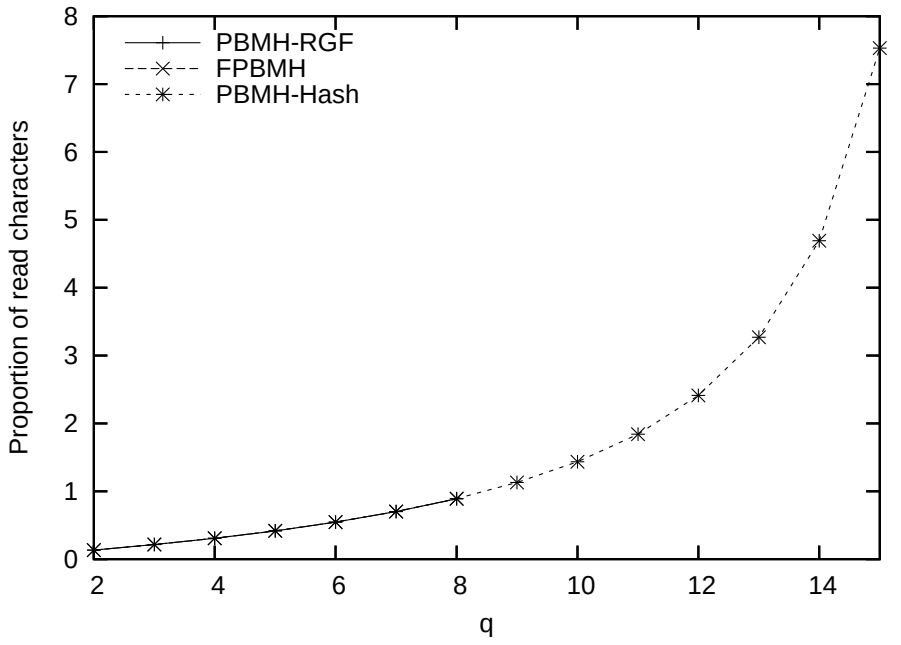
$$2q^2 + \sum_{i=0}^{\lfloor m^2/q^2 \rfloor - 3} q^2 \cdot \frac{1}{\sigma^i} = 2q^2 + q^2 \frac{\sigma}{\sigma - 1} \left(1 - \frac{1}{\sigma^{\lfloor m^2/q^2 \rfloor - 2}}\right),$$

because the pattern is $(q^2, \log_\sigma m^2)$-repetitive, and thus the predecessor string of each q-gram of the pattern contains at least one non-zero element, i.e. $z < q^2$. The work in a bad window is thus clearly $\mathcal{O}(q^2)$ if $q^2 \leq m^2/2$, which holds if $q \leq (m + 1)/2$. Because we have chosen q so that the pattern is $(q^2, \log_\sigma m^2)$-repetitive, $q^2 = \mathcal{O}(z + \log_\sigma m^2)$, and then the work in a bad window is bounded by $\mathcal{O}(\log_\sigma(m^2\sigma^z)) = \mathcal{O}(m^B s(m, \sigma, z)^B)$ for any $B > 0$. By Theorem 2.5, the average case complexity of matching the pattern against one strip of the text is $\mathcal{O}(nq^2/m)$ if $q^2 > (1 + B)(\log_\sigma m^2 + z)$ for any $B > 0$ such that $q \leq m - q + 1$, which is equal to $q \leq (m + 1)/2$. There are a total of $n/(m - q + 1) = \mathcal{O}(n/m)$ strips, and so the average complexity of the two-dimensional algorithm is $\mathcal{O}(n^2q^2/m^2)$. □

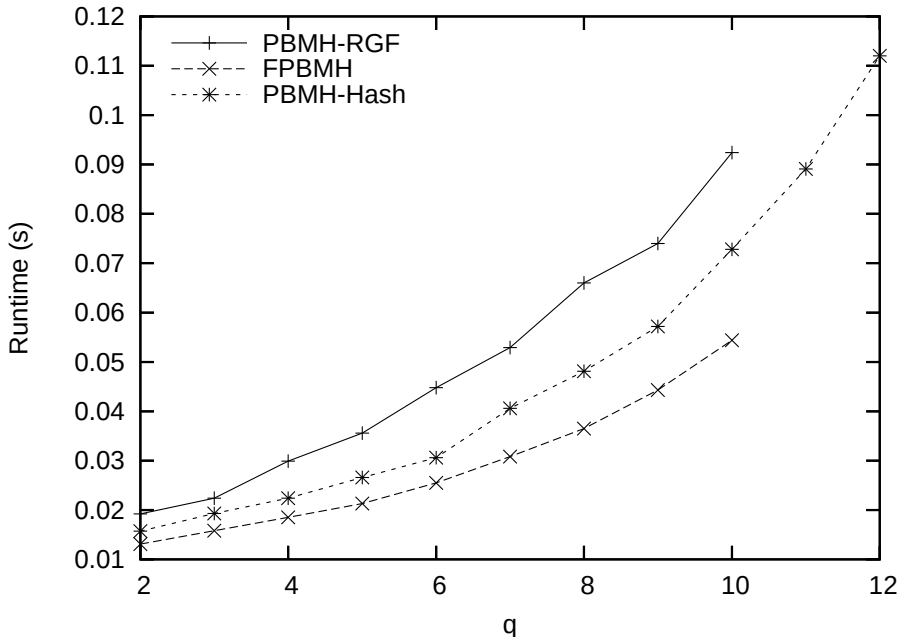
4.5 Experimental Results

The analysis predicts that the value of q should be chosen so that the pattern is $(q, \log_\sigma m)$-repetitive. To validate this, we ran our algorithms with several patterns and a randomly generated text with alphabet size 256. Figures 4.2, 4.3, and 4.4 show the proportion of read characters and the runtime for some patterns. The proportion of read characters is calculated as lookups divided by the length of the text, and thus for a sublinear algorithm, this value is less than one. The runtime does not include time used for preprocessing. All these tests were run on a computer with a 1.0 GHz AMD Athlon processor, 512 MB of memory, and 256 kB on-chip cache. The computer was running Linux 2.6.18. The algorithms were written in C and compiled with gcc 4.1.1.

Figure 4.2 shows that choosing a larger q with a highly repetitive pattern does not make the algorithms perform faster. Using 2-grams already guarantees long enough shifts, and thus assembling larger q-grams just wastes time. Figure 4.3 presents a completely different scenario. Here the pattern is not (q, ℓ)-repetitive for any q, and as can be seen, we cannot choose a large enough q to guarantee the sublinearity of the algorithms. In Figure 4.4, the situation is something in between. The pattern is (3,1)-repetitive but not (2,1)-repetitive. As can be seen, the value $q = 3$ is optimal in this situation, and using larger q-grams only makes the algorithms do more work.

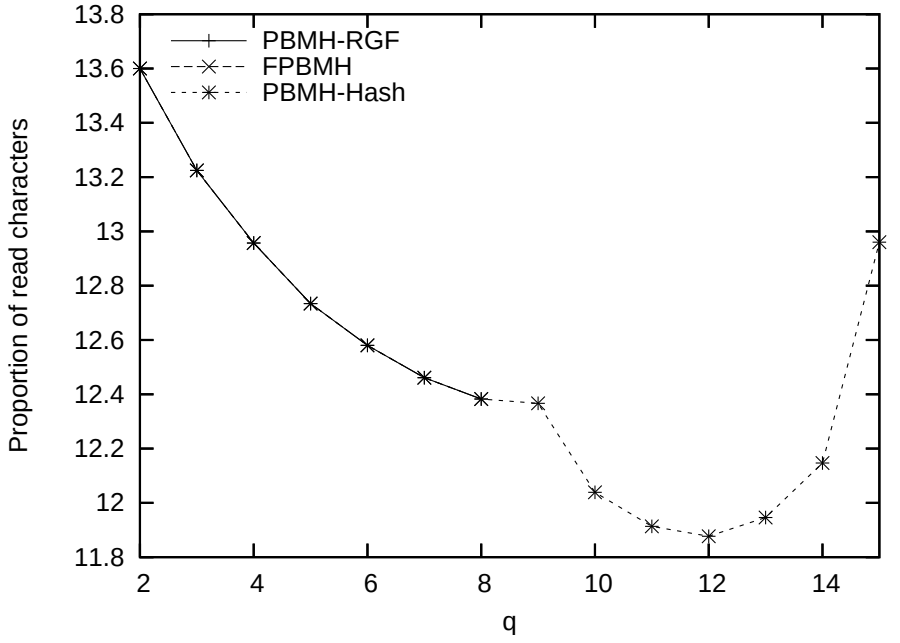


(a)

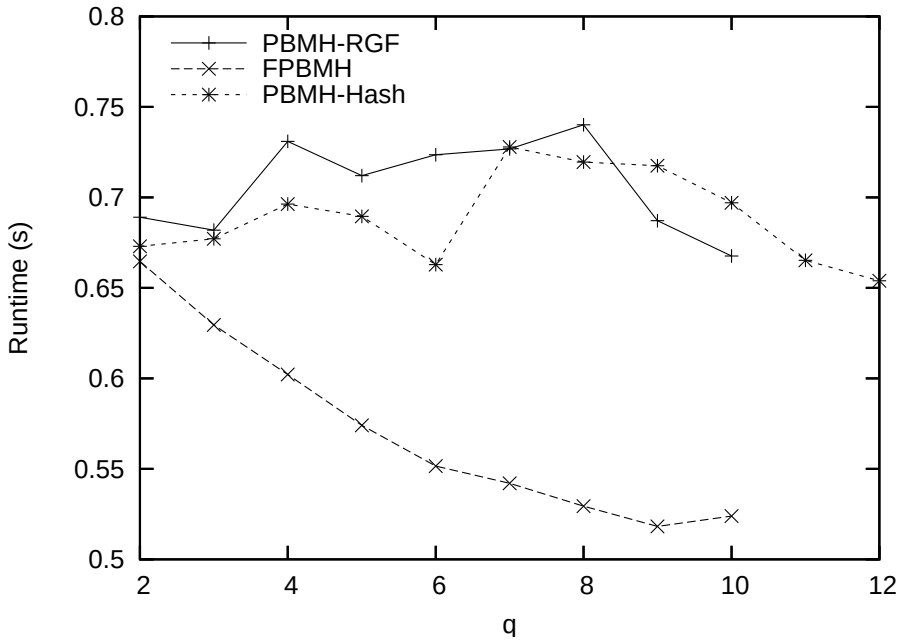


(b)

Figure 4.2: The pattern “aaaaaaaaaaaaaaa”: (a) proportion of read characters and (b) runtime in a random text

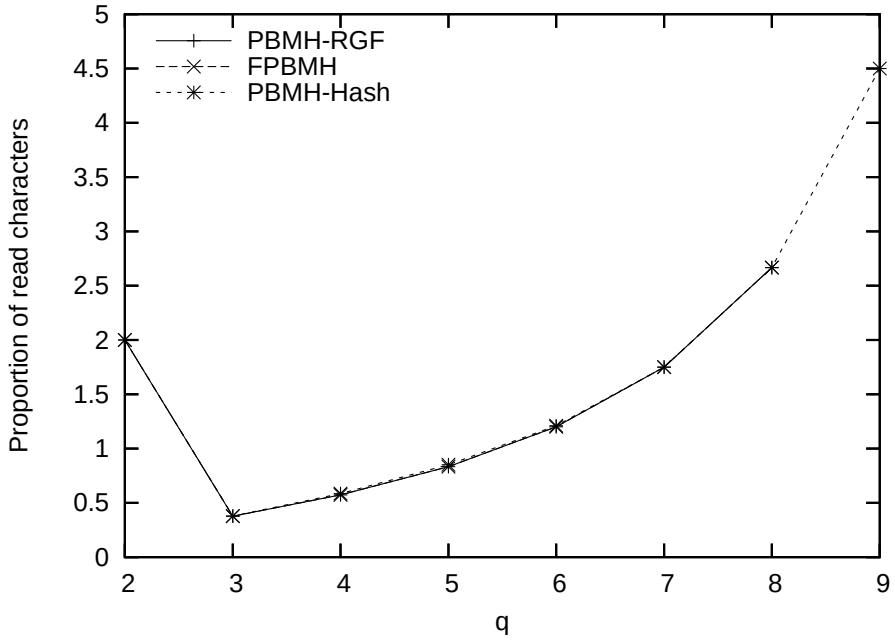


(a)

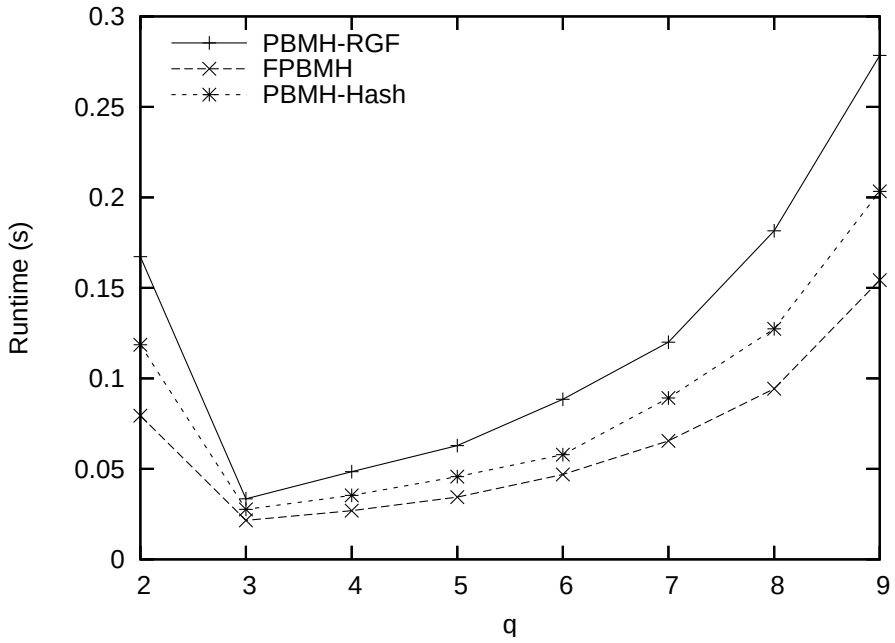


(b)

Figure 4.3: The pattern “qwertyuiopsadfgh”: (a) proportion of read characters and (b) runtime in a random text



(a)



(b)

Figure 4.4: The pattern “aassddssaa”: (a) proportion of read characters and (b) runtime in a random text

The analysis further predicts that our algorithms are sublinear on average if the pattern is $(q, \log_\sigma m)$-repetitive. To verify this, we measured the proportion of read characters on random patterns and texts with fairly small alphabet sizes. When the alphabet size is small, most of the patterns are $(q, \log_\sigma m)$-repetitive even for a fairly small q, and in fact, if we choose $q = \sigma + \ell$, all patterns are (q, ℓ)-repetitive.

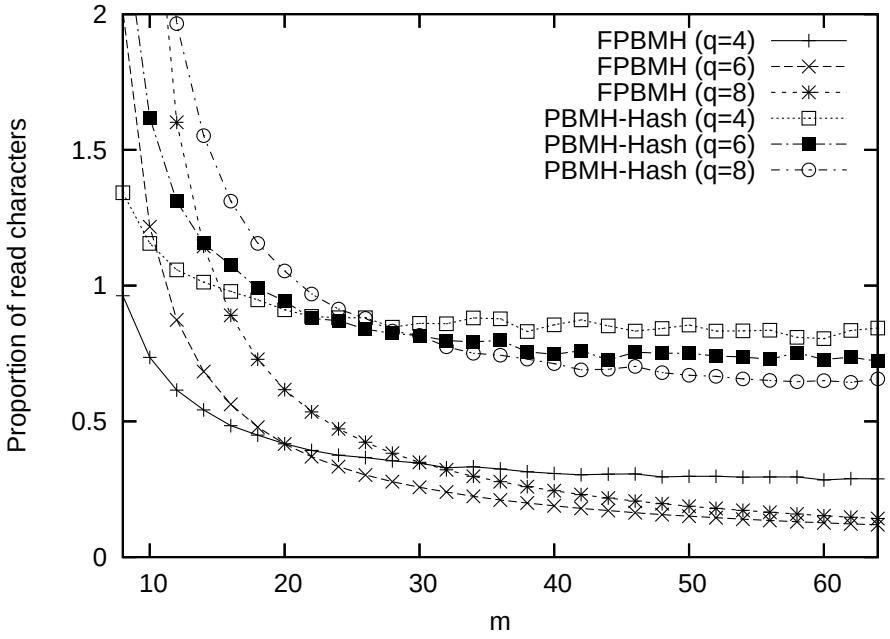
Figure 4.5 shows the results of these experiments for the FPBMH and PBMH-Hash algorithms. Because the PBMH-RGF algorithm has exactly the same shift behavior as the FPBMH algorithm, the proportion of read characters is also exactly the same. Thus, the PBMH-RGF algorithm is not included in the figure. As can be seen, the proportion of read characters falls below 1 for all the algorithms with large enough m. The PBMH-Hash algorithm performs poorer than the FPBMH algorithm in these tests because the alphabet size is quite small, which makes hash collisions more frequent. Figures 4.2(a), 4.3(a), and 4.4(a) show that with a larger alphabet, the proportion of read characters is in practice the same for PBMH-Hash and the other algorithms.

Table 4.2 shows a runtime comparison of our one-dimensional algorithms and the following algorithms:

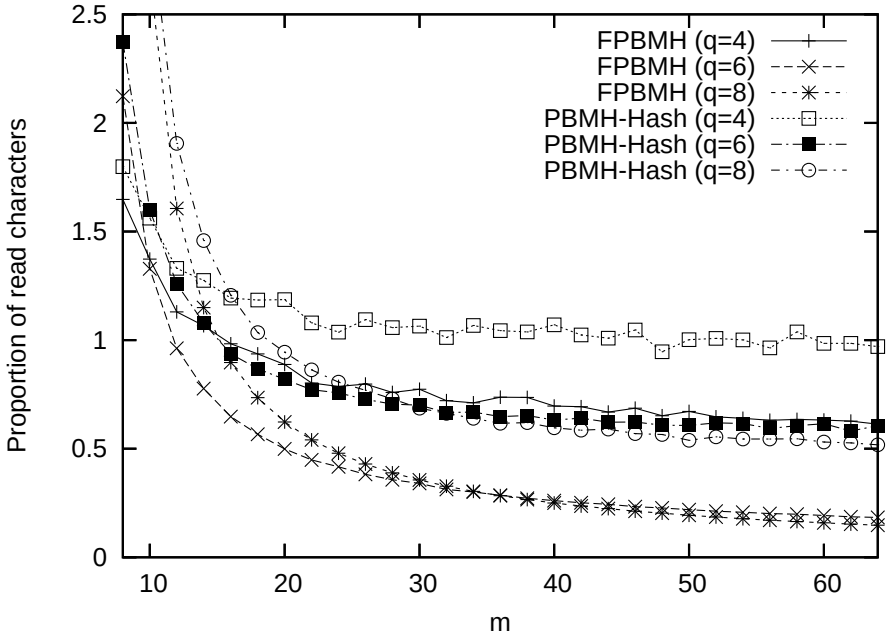
- Parameterized Boyer-Moore (PBM) by Baker [17]
- Parameterized Knuth-Morris-Pratt (PKMP) by Amir et al. [6]
- Parameterized Shift-Or (PSO) by Fredriksson and Mozgovoy [41]
- Fast Parameterized Shift-Or (FPSO) by Fredriksson and Mozgovoy [41]
- Parameterized Backward DAWG Matching (PBDM) by Fredriksson and Mozgovoy [41]

The text used in these experiments is randomly generated with alphabet size 256, and these times exclude the preprocessing time. We used a version of the PBM algorithm that only utilizes the Boyer-Moore shift rule since that turned out to be faster in practice. Our algorithms are faster when the pattern contains a substantial amount of repetition, while the linear worst case time algorithms, PSO and PKMP, are faster when there is no repetition in the pattern.

To further test our algorithms and to compare them against the other algorithms, we ran some tests with DNA data and random data with alphabet size 10. In the DNA test, the text was a chromosome from the fruit fly genome (22 MB). In both cases, the patterns were chosen randomly from the text. For those algorithms that have parameters affecting their performance (like the value of q in our algorithms), we chose the parameter values that gave the shortest running time. Figures 4.6(a) and 4.6(b) show the averages over 200 runs excluding the preprocessing time. As can be seen, our algorithms have characteristics typical to Boyer-Moore based algorithms. With longer patterns, the shifts get longer, and thus the algorithms are faster. The figures also show that the FPBMH algorithm is the fastest in both cases when the patterns are

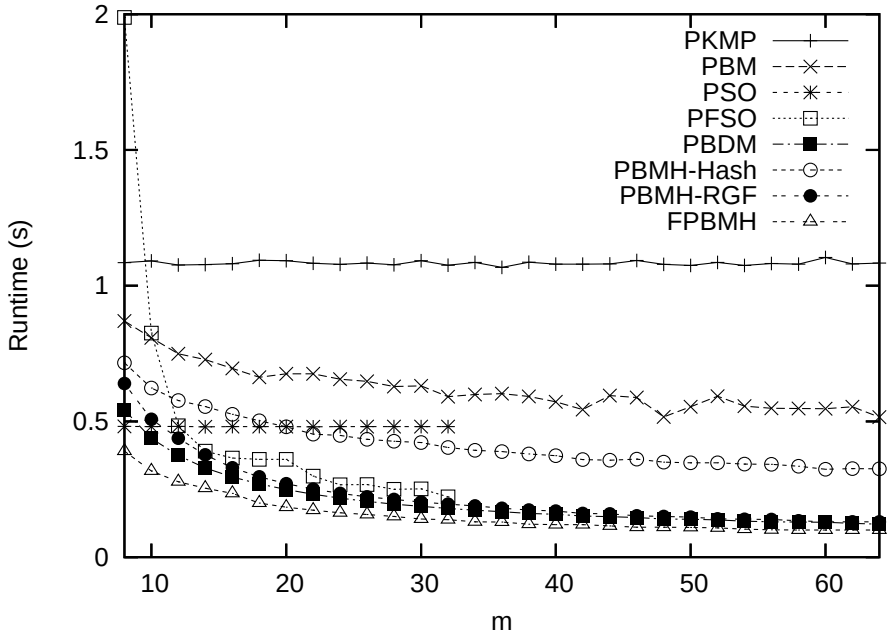


(a)

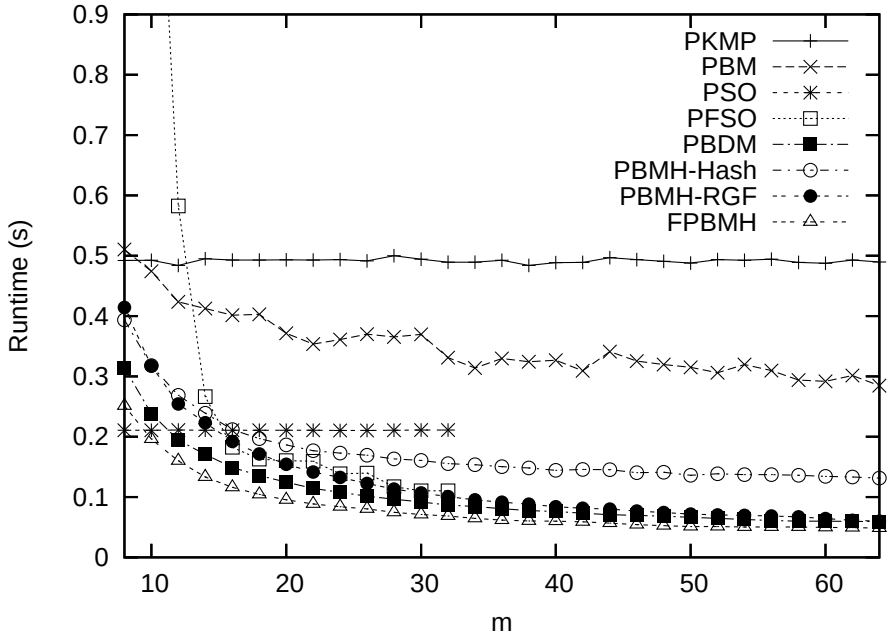


(b)

Figure 4.5: The proportion of read characters for the FPBMH and PBMH-Hash algorithms with various values of q. Text used in the experiment was (a) a chromosome from the fruit fly genome and (b) a random text with alphabet size 10.



(a) DNA



(b) Random ($\sigma = 10$)

Figure 4.6: Runtime comparison of the parameterized matching algorithms with (a) DNA data and (b) random data with alphabet size 10

Table 4.2: Runtime comparison of the one-dimensional algorithms in a random text

Algorithm	P=aaaaaaaaaaaaaaaaa	P=qwertyuiopasdfgh	P=aassddssaa
PBM	0.128 s	0.598 s	0.128 s
PKMP	0.125 s	0.141 s	0.127 s
PSO	0.065 s	0.065 s	0.065 s
FPSO	0.022 s	2.876 s	0.049 s
PBDM	0.019 s	0.841 s	0.035 s
PBMH-RGF	0.019 s	0.682 s	0.034 s
FPBMH	0.013 s	0.518 s	0.022 s
PBMH-Hash	0.016 s	0.654 s	0.028 s

Table 4.3: Proportion of read characters for two different texts and several different patterns. All the patterns are of size 8×8.

Text	Pattern: single-character	Pattern: no repetitions	Pattern: repetitions
Random	0.25	7.90	0.25
Map	1.14	0.25	0.33

at least 10 characters long. With the larger alphabet, the PSO algorithm is fastest with shorter patterns.

We ran also some tests with the two-dimensional algorithm. We used two different texts, a randomly generated text, where the characters were drawn from an alphabet of 256 characters, and a picture of a map[1] from the photo archive Gimp-Savvy.com. We examined the proportion of read characters for three different patterns of size 8×8. The first one contained repetitions of one character, the second contained no repetitions, and the third contained a map symbol with some repetition. Table 4.3 shows the results of the tests run with the two-dimensional algorithm using 3-grams. As can be seen, the algorithm performs well when the text or the pattern contains repetitions.

[1] http://gimp-savvy.com/PHOTO-ARCHIVE/UFWS/FULL/B81641997.gif

Chapter 5

Multiple String Matching with Very Large Pattern Sets

In this chapter, we consider a variation of string matching, where multiple patterns are given, and we need to find all occurrences of all the patterns. Many good solutions have been presented for this problem, e.g. Aho-Corasick [3], Commentz-Walter [29, 80], and Rabin-Karp [54, 72] algorithms with their variations. However, most of the earlier algorithms have been designed for pattern sets of moderate size, i.e. a few dozens, and they unfortunately do not scale very well to larger pattern sets. In this work, we concentrate on practical methods that can efficiently handle several thousand patterns with moderate memory usage.

We develop filtering algorithms that use q-grams to boost the filtering efficiency. Three algorithms are presented, HG, SOG, and BG, which are based on the Boyer-Moore-Horspool [49], shift-or [11], and BNDM [79] algorithms, respectively. Of these, HG and BG are q-gram backward string matching algorithms, and we prove that they are optimal on average. Wu and Manber [108] have previously used q-grams to boost a Boyer-Moore-Horspool type algorithm for multiple pattern matching, but we use q-grams in a different way to improve filtration efficiency. Related methods for a single pattern have been suggested by Fredriksson [38].

The following experimental setting was used throughout this chapter if not otherwise stated. We used a 32 MB randomly created text in the alphabet of 256 characters. Also the patterns were randomly generated in the same alphabet. The times are averages over 10 runs using the same text and patterns. Both the text and the patterns reside in the main memory in the beginning of each test in order to exclude reading times. The tests were run on a computer with a 1.0 GHz AMD Athlon dual core processor, 2 GB of memory, 64 kB L1 cache, and 512 kB L2 cache. The computer was running Linux 2.6.23. The algorithms were written in C and compiled with the gcc compiler.

5.1 Definitions

Problem 5.1. *Given a text $T = t_1 \dots t_n$ of n characters over an alphabet Σ of size σ and r patterns $P_1, \dots, P_r$ of length m in the same alphabet, the* multiple string matching problem *is to find all exact occurrences of all the patterns.*

If the lengths of the patterns are not equal, we select a substring from each pattern according to the length of the shortest pattern. We consider cases where m varies between 4 and 32 and r between 100 and 500,000.

5.2 Earlier Solutions

Many of the earlier algorithms for multiple pattern matching build a pattern trie in the preprocessing phase and use it for matching. For example, the Aho-Corasick algorithm [3], the Commentz-Walter based algorithms [29], and the Set Backward Oracle Matching (SBOM) algorithm [4] take this approach. While this works reasonably well for a small set of patterns, the memory requirements for huge pattern sets are intolerable because the trie data structure grows quite rapidly.

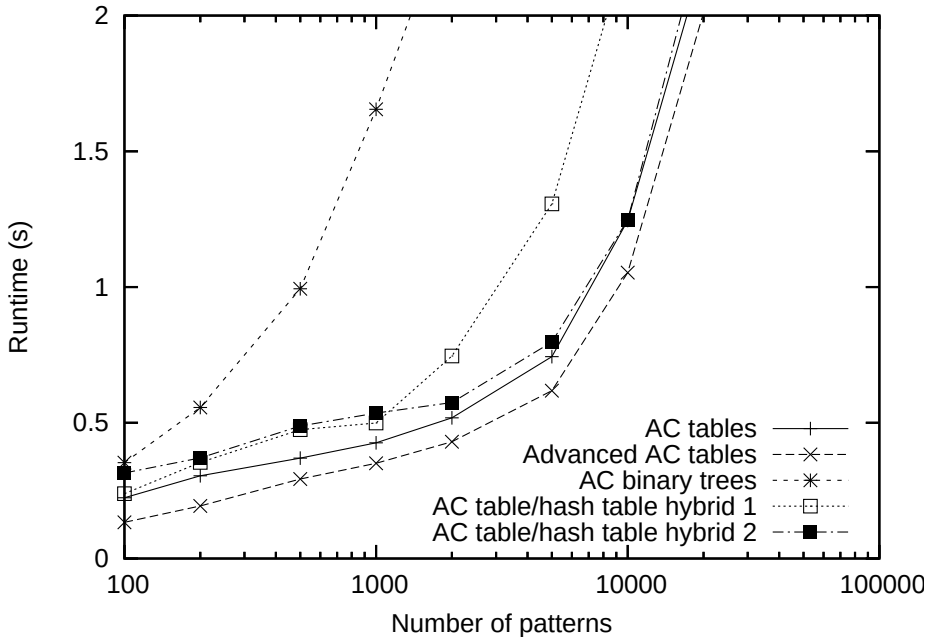
Another previous solution is to use hashing algorithms. For example, the Rabin-Karp algorithm [54] can be extended to multiple patterns. Also the Wu-Manber algorithm [108] uses hashing to extend the Boyer-Moore-Horspool algorithm [49] to multiple patterns. Another hashing approach is described in [56].

An attempt to combine the best parts of the previous solutions is described in [64]. In this solution, the pattern set is partitioned based on the length of the patterns, and then the best possible algorithm for each subset is used.

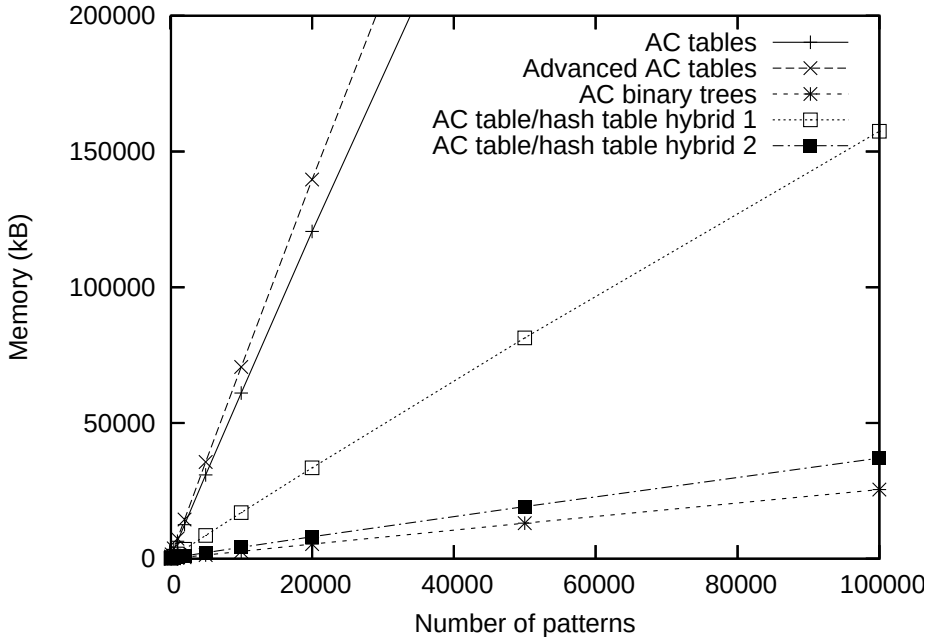
5.2.1 Aho-Corasick

The classical Aho-Corasick algorithm [3] has been widely used for multiple pattern matching. We used code based on the implementation by Fisk and Varghese [37] to test the Aho-Corasick algorithm. We tested three alternative implementations of the goto-function: table, hash table, and binary tree. The hash table version was tested with different table sizes. We also tried a combination of table and hash table implementations. In this approach, the table version was used in the first levels of the trie, while in deeper levels, the hash table implementation was utilized. We also implemented the advanced Aho-Corasick algorithm, where the failure function is incorporated into the goto function. This modification has been reported to be the fastest in practice [80]. Figure 5.1 shows the results of these experiments.

Although the speed of the Aho-Corasick algorithm is constant for small pattern sets, the situation is different for large sets even in an alphabet of moderate size. The run time graph of Figure 5.1(a) shows a steady increase. Given the memory graph of Figure 5.1(b), the hierarchical memory could explain this behavior. The advanced Aho-Corasick algorithm turned out to be the fastest also in our experiments.



(a) Runtime



(b) Memory usage

Figure 5.1: Performance of different trie implementations of the Aho-Corasick algorithm. The table/hash table hybrid 1 uses tables in the first two levels of the trie and hash tables of size 64 deeper. The second table/hash table hybrid uses tables in the first three levels and hash tables of size eight deeper.

5.2.2 Set Horspool

The Commentz-Walter algorithm [29] for multiple patterns has been derived from the Boyer-Moore algorithm [23]. A simpler variant of this algorithm is called Set Horspool [80]. (The same algorithm is called set-wise Boyer-Moore in [37].) This algorithm is developed from the Boyer-Moore-Horspool algorithm [49] for single patterns by generalizing the bad character function. The bad character function for the set of patterns is defined as the minimum of the bad character functions of individual patterns.

The reversed patterns are stored in a trie. The initial endpoint is the length of the shortest pattern. The text is compared from right to left with the trie until no matching entry is found for a character in the text. Then the bad character function is applied to the endpoint character, and the pattern trie is shifted accordingly.

We used the code of Fisk and Varghese [37] to test the Set Horspool algorithm. The same variations as for the Aho-Corasick algorithm were tried. The results on memory usage were similar to those of the Aho-Corasick algorithm because the trie structure is very similar. Also the test results on run times resemble those of the Aho-Corasick algorithm, especially with very large pattern sets. This is probably due to the memory usage.

5.2.3 Set Backward Oracle Matching

The third algorithm making use of a trie is the Set Backward Oracle Matching (SBOM) algorithm [4]. In the preprocessing phase of the SBOM algorithm, first a trie of the reversed patterns is built. Then some additional transitions are added to the trie so that at least all factors of the patterns can be recognized with the resulting factor oracle. In the matching phase, the text is scanned backward with the factor oracle. If the oracle fails to recognize a factor at a given position, we can shift the pattern beyond that position.

We ran tests on the SBOM algorithm also. The same variations for the implementation of the trie were tried. The hashing approach proved to be quite slow with SBOM because the hash tables need to have a more complex structure. In the trie built by the SBOM algorithm, a node can have several incoming links. This means that another structure is needed to implement the chaining of colliding hash table entries, while in the tries built by the AC and Set Horspool algorithms such a structure is not needed. Thus, the table implementation of the trie turned out to be the fastest.

5.2.4 Wu-Manber

The Wu-Manber algorithm [108] is a variation of the Boyer-Moore-Horspool algorithm for multiple patterns. It uses two hash tables of the last q-grams of patterns, one for determining the shift and another to locate match candidates which are verified

with pairwise comparison. Zhou et al. [112] have lately tuned the Wu-Manber algorithm for larger pattern sets by using more than one hash value of the last q-grams of the patterns and considering optimal alignments of patterns to increase the number of q-grams not appearing in any pattern in the last position.

To test the Wu-Manber algorithm, we used the code from the agrep tool [107], which is a collection of different algorithms. It uses the original Wu-Manber algorithm for exact matching of multiple patterns. We tuned the code to cope with larger pattern sets by trying larger hash tables. We tried using 2-grams without hashing, which gives the size 2^{16} for both tables. The code in the agrep tool uses hashed 3-grams. We tried four sizes for the hash tables, 2^{12}, 2^{15}, 2^{18}, and 2^{21} for determining the length of the shift and 2^{13}, 2^{16}, 2^{19}, and 2^{22} to locate the match candidates. Using 2-grams was best for small pattern sets, and using larger hash tables with hashed 3-grams was better for larger sets.

5.2.5 Rabin-Karp Approach

A well-known solution [45, 72, 114] to cope with large pattern sets with less memory is to combine the Rabin-Karp algorithm [54] with binary search. During preprocessing, hash values for all patterns are calculated and stored in an ordered table. Matching can then be done by calculating the hash value for each m-character string of the text and searching the ordered table for this hash value using binary search. If a matching hash value is found, the corresponding pattern is compared with the text. We implemented this method for $m = 8$, 16, and 32. The hash values for patterns of eight characters are calculated as follows. First, a 32-bit integer is formed of the first four bytes of the pattern and another from the last four bytes of the pattern. These are then xor'ed together resulting in the following hash function:

$$\mathrm{Hash}(s_1 \ldots s_8) = s_1 s_2 s_3 s_4 \hat{} s_5 s_6 s_7 s_8 \ .$$

The hash values for $m = 16$ and 32 are calculated in a similar fashion:

$$\begin{aligned} \mathrm{Hash16}(s_1 \ldots s_{16}) &= (s_1 s_2 s_3 s_4 \hat{} s_5 s_6 s_7 s_8) \hat{} (s_9 s_{10} s_{11} s_{12} \hat{} s_{13} s_{14} s_{15} s_{16}) \ , \\ \mathrm{Hash32}(s_1 \ldots s_{32}) &= ((s_1 s_2 s_3 s_4 \hat{} s_5 s_6 s_7 s_8) \hat{} \ldots \hat{} (s_{25} s_{26} s_{27} s_{28} \hat{} s_{29} s_{30} s_{31} s_{32})) \ . \end{aligned}$$

Muth and Manber [72] use two-level hashing to improve the performance of the Rabin-Karp method. The second hash is calculated from the first one by xor'ing together the lower 16 bits and the upper 16 bits. At preprocessing time, a bitmap of 2^{16} bits is constructed. The i:th bit is zero if no pattern has i as its second hash value and one if there is at least one pattern with i as its second hash value. When matching, one can quickly check from the bit table when the first hash value does not need further inspection and thus avoid the time consuming binary search in many cases. In the following, we use the shorthand RKBT for the Rabin-Karp algorithm combined with binary search and two-level hashing.

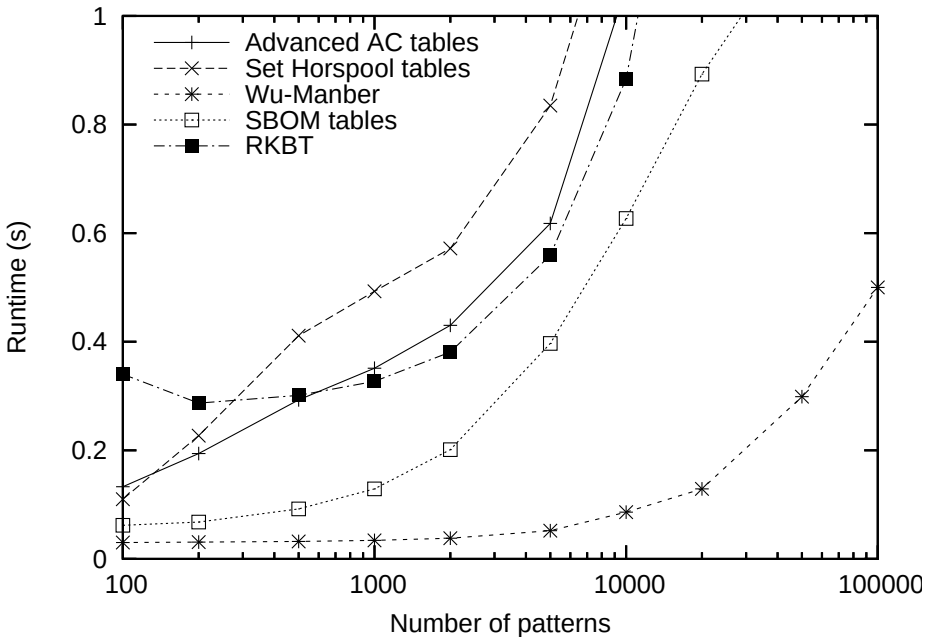


Figure 5.2: Run time comparison of the earlier algorithms

The Rabin-Karp approach was tested both with and without two-level hashing. The use of the second hash table of 2^{16} bits significantly improves the performance of the algorithm when the number of patterns is less than 100,000. When there are more patterns, a larger hash table should be considered because this hash table tends to be full of ones and the gain of two-level hashing disappears.

5.2.6 Comparison of the Earlier Algorithms

Figure 5.2 shows a comparison of the earlier algorithms. The times do not include preprocessing. The run times of the Wu-Manber algorithm are the best ones obtained by different sizes of the hash tables. In the experiments of Navarro and Raffinot [80], Wu-Manber was the fastest algorithm for 1,000 patterns for $m = 8$, which holds true also for our experiment.

5.3 Filtering Algorithms

A filtering method is able to determine fast that a position does not match, for example, by figuring out that a substring of the window does not match any of the patterns. However, the candidate matches returned by a filtering method can be false positives, and thus these must be verified later to determine the true matches.

In this work, we develop filtering algorithms for multiple pattern matching. All of our algorithms operate in three phases. The pattern is first preprocessed, in the second phase we search the text with a filtering method, and the candidate matches produced by the filtering are verified in the third phase.

All of our algorithms can be viewed as character class filters although the generalized pattern with character classes is not explicitly built. A character class filter algorithm first builds a generalized pattern containing character classes which matches all of the patterns in the pattern set. For example, if the pattern set is {"pattern", "filters"}, then the corresponding generalized pattern is [f,p][a,i][l,t][t][e][r][n,s]. This generalized pattern is then given to a single pattern algorithm able to handle character classes to generate the candidate matches. Given this scheme, it is obvious that all actual occurrences of the patterns will be reported by the filter, but there are also false positives. In the previous example, "falters" is an example of a false positive.

When the pattern set grows, the filtering efficiency of the above scheme starts to deteriorate as the character classes in each position contain almost all characters. To counter this problem, the patterns can first be transformed into sequences of overlapping q-grams. For example if we utilized 2-grams, the above pattern set would become {"pa-at-tt-te-er-rn", "fi-il-lt-te-er-rs"}, and the generalized pattern would be [fi,pa][at,il][lt,tt][te][er][rn,rs].

A filtering algorithm always requires an exact algorithm to verify the candidate matches. In principle, any of the presented earlier methods could be used for this purpose. A trie-based method is fast if the trie does not grow too large. With large pattern sets, we can limit the depth of the trie to control memory requirements. Another possibility is to use the RKBT method, which has very modest memory requirements, but the binary search can be time consuming. We have implemented our methods by using RKBT to verify the candidate matches.

5.3.1 Multi-Pattern Shift-Or with q-Grams

The shift-or algorithm is easily extended to handle classes of characters in the pattern [2, 11], and thus developing a filtering algorithm for multiple pattern matching is straightforward. The preprocessing phase now initializes the bit vectors for each character in the alphabet as follows. The i:th bit is set to 0 if the given character is included in the character class in the i:th position. Otherwise the bit is set to 1. The filtering phase proceeds then exactly like the matching phase of the shift-or algorithm. Given this scheme, it is clear that all actual occurrences of the patterns in the text are candidates. However, there are also false positives as the generalized pattern matches also other strings than the original patterns. Therefore, each candidate must be verified with the RKBT method.

When the number of patterns grows, this approach is no longer adequate as the generalized pattern accepts almost all characters in each position. The filtering capability can then be considerably improved by utilizing q-grams. The patterns are transformed to sequences of $m - q + 1$ overlapping q-grams, i.e. a q-gram starts at each position

of the patterns and we only consider those q-grams that are fully inside the patterns. The bit vectors are initialized for each q-gram, and so the i:th bit is 0 if the q-gram occurs in any of the patterns starting at position i, and otherwise the i:th bit is set to 1. In the filtering phase, we read overlapping q-grams from the text and use the q-grams to index the tables. Note that the next overlapping q-gram can be computed from the previous q-gram and the next character of the text in $\mathcal{O}(1)$ time. A candidate match has been found if the $m-q+1$:th bit of the state vector is 0. We call our modification SOG (short for Shift-Or with q-Grams). The improved efficiency of this approach is achieved at the cost of space. The bit vectors will now take $m\sigma^q$ bits space.

Baeza-Yates and Gonnet [11] present a way to extend the shift-or algorithm for multiple patterns for small values of r. Patterns $P_1 = p_1^1 \ldots p_m^1, \ldots, P_r = p_1^r \ldots p_m^r$ are concatenated into a single pattern:

$$P = p_1^1 p_1^2 \ldots p_1^r p_2^1 p_2^2 \ldots p_2^r \ldots p_m^1 p_m^2 \ldots p_m^r \ .$$

The patterns can then be searched in the same way as a single pattern, except that the shift of the state vector will be for r bits, and a match is found if any of the r bits corresponding to the highest positions is 0. This method can also be applied to SOG to make the algorithm faster for short patterns. The pattern set is divided into four or two subsets based on the first q-gram. Each subset is then transformed into a general pattern like in the plain SOG algorithm. The extension method of Baeza-Yates and Gonnet is then applied to these general patterns.

The above organization for P is convenient in the sense that when the state vector is shifted r bits to the left, the r rightmost bits are automatically initialized to zero. However, we could also think of organizing the characters in P slightly differently:

$$P = p_1^1 p_2^1 \ldots p_m^1 p_1^2 p_2^2 \ldots p_m^2 \ldots p_1^r p_2^r \ldots p_m^r \ .$$

With this organization, we need to shift the state vector one bit to the left, clear the bits corresponding to the first characters of the patterns, and a match is found if any of the bits corresponding to the last characters of the patterns are zero. This approach is more cumbersome than the previous approach if all patterns are of the same length. However, if the patterns have varying lengths, the latter definition of P allows for more efficient utilization of the bits in a computer word. In this case, we can partition the pattern set according to the length of the patterns. For example, we could have one set for patterns of length 4 to 7, one set for patterns of length 8 to 12, and so on.

Fredriksson and Grabowski [39, 40] have proposed a modification to enhance the performance of the shift-or algorithm. In their scheme, several patterns are formed from the original one by taking every k:th character starting at different offsets. For example, for $k = 2$ the pattern 'pony' would produce patterns 'pn' and 'oy'. Now we can scan the text reading every k:th character and use the shift-or algorithm to find likely matches. These candidates can then be verified. We tried this modification for SOG, but the shorter patterns produced more spurious hits, and the scanning is a bit more complicated. Thus, this modification did not make SOG faster.

5.3.2 Multi-Pattern BNDM with q-Grams

Our second filtering algorithm is based on the BNDM algorithm by Navarro and Raffinot [79]. This algorithm has been extended to classes of characters in the same way as the shift-or algorithm. We call the resulting multiple pattern filtering algorithm BG (short for BNDM with q-Grams). The bit vectors of the BNDM algorithm are initialized in the preprocessing phase so that the i:th bit is 1 if the corresponding character is included in the character class of the reversed generalized pattern in position i. In the filtering phase, the matching is then done with these bit vectors. As with SOG, all match candidates reported by this algorithm must be verified. The verification phase of the algorithm uses the RKBT method.

Just like in SOG, q-grams can be used to improve the efficiency of the filtering. That is, the pattern is transformed into a string of q-grams, the bit vectors are initialized for each q-gram rather than for a single character, and the text is read one q-gram at a time. Also the division to subsets, presented for the SOG algorithm, can be used with the BG algorithm although with variable length patterns the gain is not so good as in SOG as the maximum shift length will still be limited by the shortest pattern. This scheme works in the same way as with SOG algorithm, except that the subsets are formed based on the last q-gram of the patterns.

5.3.3 Multi-Pattern Horspool with q-Grams

The last of our algorithms uses a Boyer-Moore-Horspool [49] type method for matching the generalized pattern against the text. Strictly speaking, this algorithm does not handle character classes properly. It will return all those positions where the generalized pattern matches and also some others. This algorithm is called HG (short for Horspool with q-Grams).

The preprocessing phase of HG constructs a bit table for each of the m pattern positions. The first table keeps track of characters contained in the character class of the first position of the generalized pattern, the second table keeps track of characters contained in the character classes of the first and the second position in the generalized pattern, and so on. Finally, the m:th table keeps track of characters contained in any of the character classes of the generalized pattern. Figure 5.3(a) shows the six tables corresponding to the pattern 'qwerty'.

These tables can then be used in the filtering phase as follows. First, the m:th character is compared with the m:th table. If the character does not appear in this table, the character cannot be contained in the character classes of positions $1 \ldots m$ in the generalized pattern, and a shift of m characters can be made. If the character is found in this table, the $m-1$:th character is compared to the $m-1$:th table. A shift of $m-1$ characters can be made if the character does not appear in this table and therefore not in any character class in the generalized pattern in positions $1, \ldots, m-1$. This process is continued until the algorithm has advanced to the first table and found a match candidate there. The pseudo code is shown in Figure 5.3(b). Given this

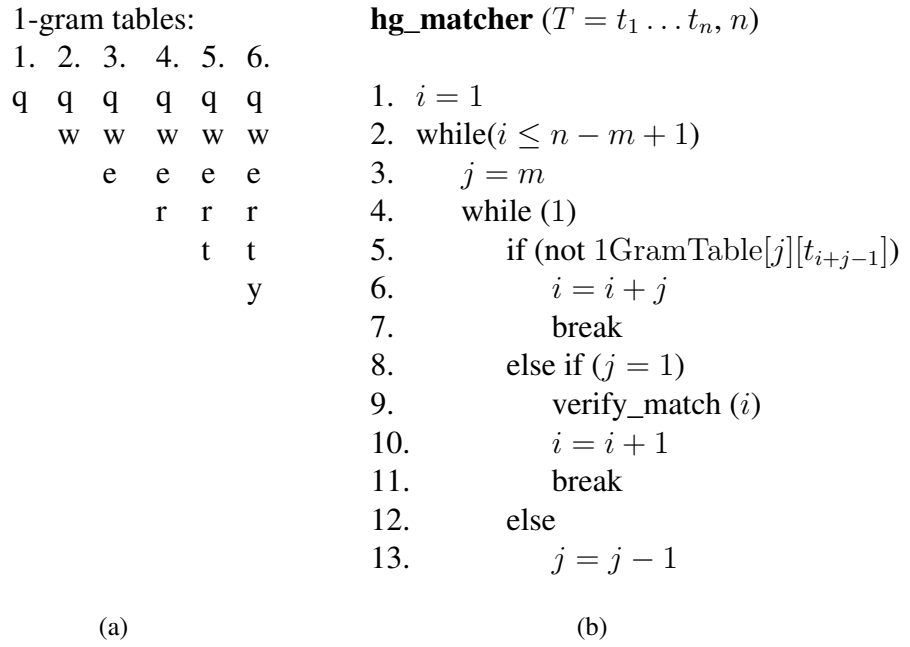


Figure 5.3: The HG algorithm: (a) the data structures for the pattern 'qwerty' and (b) the pseudo code for the search phase.

procedure, it is clear that all positions matching the generalized pattern are found. However, also other strings will be reported as matches. For example, 'qqqqqq' is a false candidate in the example of Figure 5.3(a). In the verification phase, the candidates are verified by using the RKBT method described in Section 5.2.5. As with SOG and BG, the filtering efficiency of HG can be considerably improved with large pattern sets by utilizing q-grams.

5.4 Analysis

Let us consider the time complexities of the new algorithms, HG, SOG, and BG. The algorithms can be divided into three phases: preprocessing, filtering, and verification. When considering the average case complexity, we assume the standard random string model, where each character of the text and the patterns is selected uniformly and independently at random.

All of our algorithms use the RKBT method for the verification phase. In the best case, no match candidates are found, and then checking needs no time. In the worst case, there are $n - m + 1 = \mathcal{O}(n)$ candidates, and all the patterns and text positions have the same hash value. In this case, we need to inspect the text pairwisely with each pattern, and the worst case time complexity is thus $\mathcal{O}(nrm)$. If we assume that all

patterns produce different hash values, the worst case complexity is $\mathcal{O}(n(\log r + m))$, where $\mathcal{O}(\log r)$ comes from the binary search and $\mathcal{O}(m)$ from pairwise inspection.

The preprocessing phase of the filtering phases of the three algorithms is similar, and it works in $\mathcal{O}(rm)$ in BG and SOG and in $\mathcal{O}(rm^2)$ in HG, as HG sets $\mathcal{O}(m)$ bits for each of the $\mathcal{O}(rm)$ q-grams of the patterns, while BG and SOG only set one bit per q-gram. Additionally, the initialization of the descriptor bit vectors needs $\mathcal{O}(\sigma^q)$. The preprocessing of the verification phase consists of calculating the hash values of the patterns and sorting the patterns according to these values. The sorting of the patterns takes $\mathcal{O}(r \log r)$. [1]

Let us first consider the filtering phase of SOG. We assume that $m \leq w$ holds, where w is the word length of the computer. Furthermore, we consider the time complexity of SOG without division to subsets.

Theorem 5.2. *On average the combined cost of filtering and verification in SOG is $\mathcal{O}(n)$ if we choose q so that*

$$q \geq \frac{m \log_\sigma r}{m - \log_\sigma(m + \log r)} \ .$$

Proof. In SOG, the filtering phase is clearly linear with respect to n. The probability that a random q-gram matches a given position in any of the patterns is at most r/σ^q because there are σ^q different q-grams, and at most r of these can appear in the given position in at least one of the patterns. Thus, the number of candidates in SOG is

$$C_q \leq (n - m + 1) \left(\frac{r}{\sigma^q}\right)^{\lfloor m/q \rfloor} < n \left(\frac{r}{\sigma^q}\right)^{\lfloor m/q \rfloor} \ .$$

Note that this estimate considers only those q-grams which do not overlap. Thus, the real number of candidates is lower. The average complexity of verification in SOG is thus

$$n \left(\frac{r}{\sigma^q}\right)^{\lfloor m/q \rfloor} \cdot \mathcal{O}(m + \log r) \ . \tag{5.1}$$

If this complexity is $\mathcal{O}(n)$, then the combined complexity of filtering and verification in SOG is also linear. This will surely be the case if

$$\left(\frac{r}{\sigma^q}\right)^{\lfloor m/q \rfloor} (m + \log r) \leq 1 \ .$$

For the sake of this analysis, we will now assume that q divides m so that $\lfloor m/q \rfloor = m/q$. Note that the analysis could then be extended to hold for any $m \geq q$ because we can choose an $m' < m$ such that q divides m' and use pattern prefixes of length m' for

[1] Our current implementation utilizes the Quicksort algorithm, which runs in $\mathcal{O}(r^2)$ time in the worst case and in $\mathcal{O}(r \log r)$ time in the average case.

the filtering phase. By taking logarithms of both sides of the above equation, we get

$$\begin{aligned}\frac{m}{q}(\log_\sigma r - q) + \log_\sigma(m + \log r) &\leq 0 \\ \frac{m}{q}\log_\sigma r &\leq m - \log_\sigma(m + \log r) \\ q &\geq \frac{m \log_\sigma r}{m - \log_\sigma(m + \log r)} \quad .\end{aligned}$$

Therefore, the verification cost of SOG is $\mathcal{O}(n)$ on average if q is chosen according to the above equation, which completes the proof. □

With the above choice of $q = \Theta(\log_\sigma r)$, the space complexity of SOG is $\mathcal{O}(\sigma^q + rm) = \mathcal{O}(r^{\Theta(1)} + rm)$, which includes also structures for verification. Similarly, the time complexity of preprocessing including initialization of the descriptor bit vectors and preprocessing for verification is $\mathcal{O}(rm + \sigma^q + r \log r) = \mathcal{O}(r \cdot \max(m, \log r) + r^{\Theta(1)})$.

Let us then consider the filtering phase in BG and HG. For BG, we assume that $m \leq w$ holds, where w is the word length of the computer, and we consider the time complexity of BG without division to subsets. The worst case complexity of filtering in both BG and HG is $\mathcal{O}(mn)$ because in the worst case both algorithms always read the whole window of m characters and always shift the pattern by one position.

The following theorem establishes the average case complexity of filtering for both HG and BG.

Theorem 5.3. *If $q = c\log_\sigma(rm) \leq m/2$ for a constant $c > 1$, then the average case complexity of filtering in BG and HG is $\mathcal{O}(n \log_\sigma(rm)/m)$. The analysis is valid for $r < \sigma^{\frac{m}{2}}/m$.*

Proof. The filtering phase in both BG and HG is a q-gram backward string matching algorithm as defined in Section 2.4. The length of a q-gram is clearly q, and so $g(q) = q$. The probability that a random q-gram matches any of the patterns in any position is at most rm/σ^q, because there are σ^q different q-grams, and less than rm of these can occur in the patterns. This is also the probability that a window is bad, and so $s(r) = r$ and $A = 1$. Clearly both algorithms read $\mathcal{O}(q)$ characters in a good window and make a shift of length $f(m, q) = m - q + 1$ after that.

If the window is bad, both algorithms will read some of the previous q-grams to determine if there is a potential match. Both algorithms will stop if they encounter a q-gram that does not occur in any of the patterns. In the worst case, the last q characters match because this is a bad window, and the previous q characters match because of the previous shift. The average number of characters read by the algorithms is thus at most

$$2q + \sum_{i=0}^{\lfloor m/q \rfloor - 3} q \cdot \left(\frac{rm}{\sigma^q}\right)^i = 2q + q \cdot \frac{\sigma^q}{\sigma^q - rm}\left(1 - \left(\frac{rm}{\sigma^q}\right)^{\lfloor m/q \rfloor - 2}\right) \quad ,$$

which is asymptotically $\mathcal{O}(q)$ if $c\log_\sigma(rm) \leq q \leq m/2$, where $c > 1$ is a constant. Note that this estimate is conservative as we are considering only those q-grams that are independent, i.e. do not overlap. If we choose $q = c\log_\sigma(rm) = \Theta(\log_\sigma(rm))$, then the work done in bad windows is $\mathcal{O}(q) = \mathcal{O}(\log_\sigma(rm)) = \mathcal{O}(s(r)^B m^B)$ for any $B > 0$. By Theorem 2.5, the filtering in HG and BG is thus $\mathcal{O}(nq/(m-q+1)) = \mathcal{O}(n\log_\sigma(rm)/m)$ if $q > (B+1)\log_\sigma(rm)$ for any constant $B > 0$ such that $q \leq m - q + 1$. The condition $q \leq m - q + 1$ is equal to $q \leq (m+1)/2$, which always holds if $q \leq m/2$. Such a q can be found if $\log_\sigma(rm) < m/2$ or equally if $r < \sigma^{\frac{m}{2}}/m$. □

Navarro and Fredriksson [77] have shown that the lower bound for the average complexity of multiple string matching is $\Omega(n\log_\sigma(rm)/m)$, and the following theorems prove that with an appropriate choice of q, both BG and HG are average optimal.

Theorem 5.4. *The complexity of filtering and verification in BG is $\mathcal{O}(n\log_\sigma(rm)/m)$ on average if we choose $q = c\log_\sigma(rm)$ for a constant $c > 1$ such that*

$$\frac{m\log_\sigma r}{m + \log_\sigma\log_\sigma(rm) - \log_\sigma(m+\log r) - \log_\sigma m} \leq q \leq m/2 \ .$$

The analysis is valid for $r < \sigma^{\frac{m}{2}}/m$.

Proof. Theorem 5.3 shows that if $r < \sigma^{\frac{m}{2}}/m$, the cost of filtering in BG is $\mathcal{O}(n\log_\sigma(rm)/m)$ if $q = c\log_\sigma(rm)$ for a constant $c > 1$ such that $q \leq m/2$. It remains to show that the verification cost of BG is also bounded by $\mathcal{O}(n\log_\sigma(rm)/m)$.

The expected number of candidates for the BG algorithm is the same as for the SOG algorithm, and so the complexity of the verification phase of BG is also given by Equation 5.1. However, filtering in BG is sublinear on average so we need a stricter condition to assure that the complexity of verification phase is not higher than the complexity of filtering. Clearly the complexity of the verification phase will be bounded by $\mathcal{O}(n\log_\sigma(rm)/m)$ if

$$\left(\frac{r}{\sigma^q}\right)^{\lfloor m/q \rfloor}(m + \log r) \leq \frac{\log_\sigma(rm)}{m} \ .$$

To simplify this analysis, we will again assume that q divides m. When we take logarithms on both sides, we get

$$\begin{aligned}
\frac{m}{q}(\log_\sigma r - q) + \log_\sigma(m+\log r) &\leq \log_\sigma\log_\sigma(rm) - \log_\sigma m \\
\frac{m}{q}\log_\sigma r &\leq m + \log_\sigma\log_\sigma(rm) - \log_\sigma(m+\log r) - \log_\sigma m \\
q &\geq \frac{m\log_\sigma r}{m + \log_\sigma\log_\sigma(rm) - \log_\sigma(m+\log r) - \log_\sigma m} \ .
\end{aligned}$$

So if we choose q according to the above equation and Theorem 5.3, then the average complexity of filtering and verification in BG is $\mathcal{O}(n\log_\sigma(rm)/m)$. □

Theorem 5.5. *The complexity of filtering and verification in HG is* $\mathcal{O}(n \log_\sigma(rm)/m)$ *on average if* $q = c \log_\sigma(rm)$ *for a constant* $c > 1$ *such that*

$$\frac{m \log_\sigma(rm)}{m + \log_\sigma \log_\sigma(rm) - \log_\sigma(m + \log r) - \log_\sigma m} \leq q \leq m/2 \ .$$

The analysis is valid for $r < \sigma^{\frac{m}{2}}/m$.

Proof. Theorem 5.3 shows that if $r < \sigma^{\frac{m}{2}}/m$ then the cost of filtering in HG is $\mathcal{O}(n \log_\sigma(rm)/m)$ if $q = c \log_\sigma(rm)$ for a constant $c > 1$ such that $q \leq m/2$. Thus, it remains to show that the verification cost of HG is also bounded by $\mathcal{O}(n \log_\sigma(rm)/m)$.

The probability of a q-gram appearing in the j:th q-gram table is at most rj/σ^q because there are σ^q different q-grams and at most rj of these have been added to the j:th table at the preprocessing phase. The probability of finding a candidate in the HG algorithm is the probability that each q-gram in the window is found in the corresponding q-gram table. Thus, the expected number of candidates is

$$\begin{aligned} C_q &< (n - m + 1) \prod_{j=1}^{\lfloor m/q \rfloor} \frac{rj + r(q-1)(j-1)}{\sigma^q} \\ &< n \left(\frac{rm}{\sigma^q} \right)^{\lfloor m/q \rfloor} \ . \end{aligned}$$

Note that as with SOG and BG, this estimate considers only those q-grams which do not overlap, and therefore the real number of candidates is lower. The average complexity of verification in HG is thus

$$n \left(\frac{rm}{\sigma^q} \right)^{\lfloor m/q \rfloor} \cdot \mathcal{O}(m + \log r) \ .$$

The verification cost is clearly bounded by $\mathcal{O}(n \log_\sigma(rm)/m)$ if

$$\left(\frac{rm}{\sigma^q} \right)^{\lfloor m/q \rfloor} (m + \log r) \leq \frac{\log_\sigma(rm)}{m} \ .$$

We will again assume that q divides m. By taking logarithms on both sides of the previous equation, we get

$$\begin{aligned} \frac{m}{q}(\log_\sigma(rm) - q) + \log_\sigma(m + \log r) &\leq \log_\sigma \log_\sigma(rm) - \log_\sigma m \\ \frac{m}{q} \log_\sigma(rm) &\leq m + \log_\sigma \log_\sigma(rm) - \log_\sigma(m + \log r) - \log_\sigma m \\ q &\geq \frac{m \log_\sigma(rm)}{m + \log_\sigma \log_\sigma(rm) - \log_\sigma(m + \log r) - \log_\sigma m} \ . \end{aligned}$$

So if we choose q according to the above equation and Theorem 5.3, then the average complexity of filtering and verification in HG is $\mathcal{O}(n \log_\sigma(rm)/m)$. □

With the above choice of $q = \Theta(\log_\sigma(rm))$, the space complexity of BG and HG is $\mathcal{O}(\sigma^q + rm) = \mathcal{O}((rm)^{\Theta(1)})$, which includes also structures for verification. Similarly, the time complexity of preprocessing including initialization of the descriptor bit vectors and preprocessing for verification is $\mathcal{O}(rm + \sigma^q + r \log r) = \mathcal{O}(r \log r + (rm)^{\Theta(1)})$ in BG and $\mathcal{O}(rm^2 + \sigma^q + r \log r) = \mathcal{O}(r \cdot \max(m^2, \log r) + (rm)^{\Theta(1)})$ in HG.

5.5 Experiments

We tested the new algorithms with various values of q. For alphabet size 256, the 2-gram versions optimize the reading of a 2-gram by using a single instruction to fetch a halfword from memory [38].

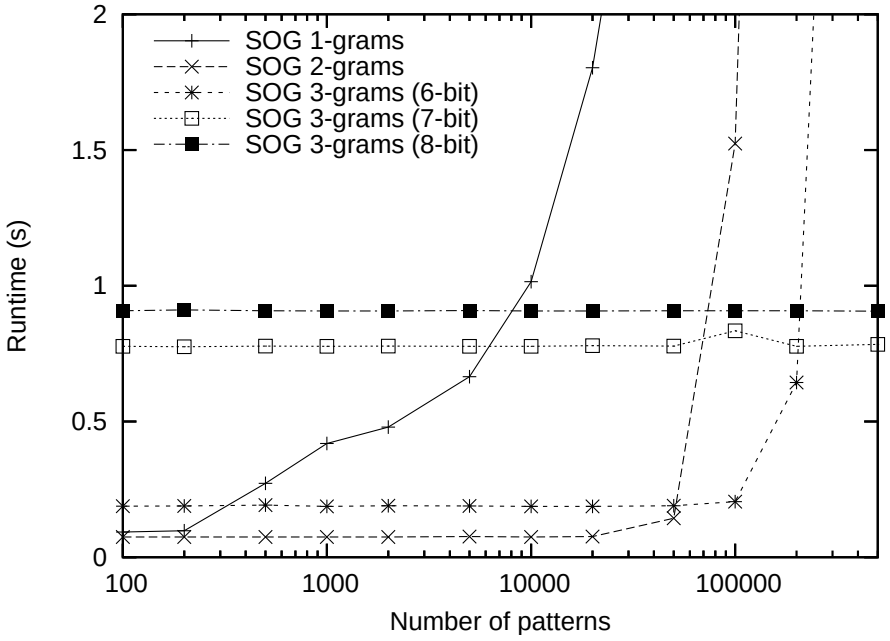
5.5.1 SOG Algorithm

The analysis predicts that we should use $q = \Theta(\log_\sigma r)$ in the SOG algorithm to assure linear running time. To verify this, we ran tests with different values of q for the standard test setting with alphabet size 256, and to get more fine grained results, we also tried the algorithm with alphabet size 4. In the latter test, the text used was a chromosome from the fruit fly genome (22 MB), and the patterns of length 32 were randomly generated. The time to preprocess the patterns is not included in the runtime of the algorithm.

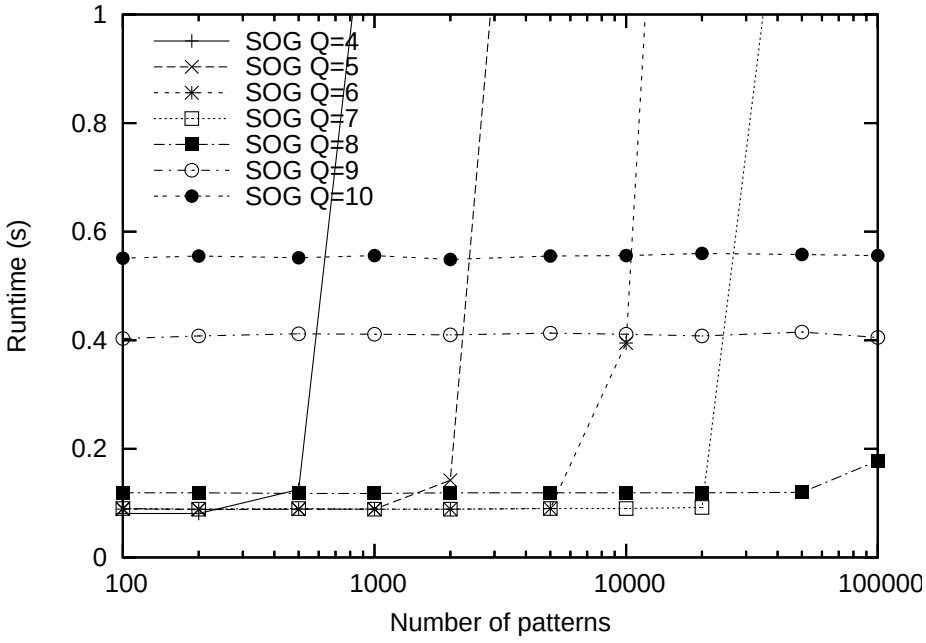
When we switch from 2-grams to 3-grams with alphabet size 256, the memory used by the structure of SOG grows from 64 kB to 16 MB. This slows the algorithm down considerably because a 16 MB table does not fit into the cache of the processor. To alleviate this problem, we also tried hashing the characters to 6- or 7-bit values before forming the 3-gram.

The results of these test are shown in Figure 5.4. The analysis predicts that with $\sigma = 256$ and $m = 8$, 1-grams suffice for 100 patterns and 2-grams until about 20,000 patterns. After that 3-grams should be used. The results of the experiments confirm nicely to these analytical results, as the 1-gram version is as fast as the 2-gram version until 200 patterns, the 2-gram version has a constant running time until 20,000 patterns, and the runtime of the 3-gram version stays constant for all tested values of r. The results with $\sigma = 4$ are similar. Figure 5.4(b) shows that the minimum applicable value for q is clearly proportional to $\log r$.

To test the sensitivity of SOG to other parameters, we ran some more tests with alphabet size 256 using 2-grams, as the 2-gram version performs reasonably well until 100,000 patterns. First, SOG was tested with pattern lengths $m = 8$, 16, and 32, see Figure 5.5(a). The figure shows the algorithm is slower for longer patterns. The structures of the SOG algorithm take 64 kB memory for $m = 8$, 128 kB for $m = 16$, and 256 kB for $m = 32$. The increased memory usage seems to slow down the algorithm.

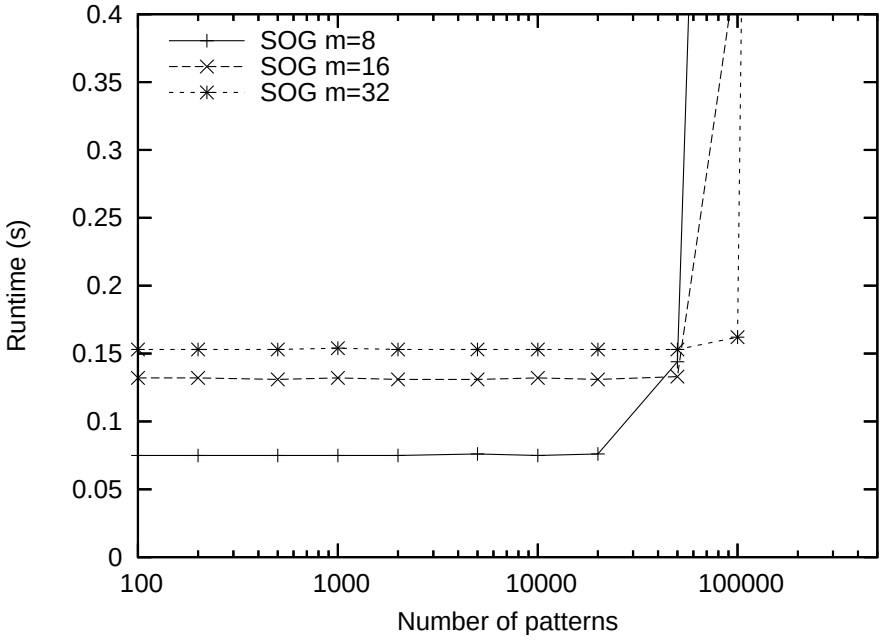


(a) $\sigma = 256$

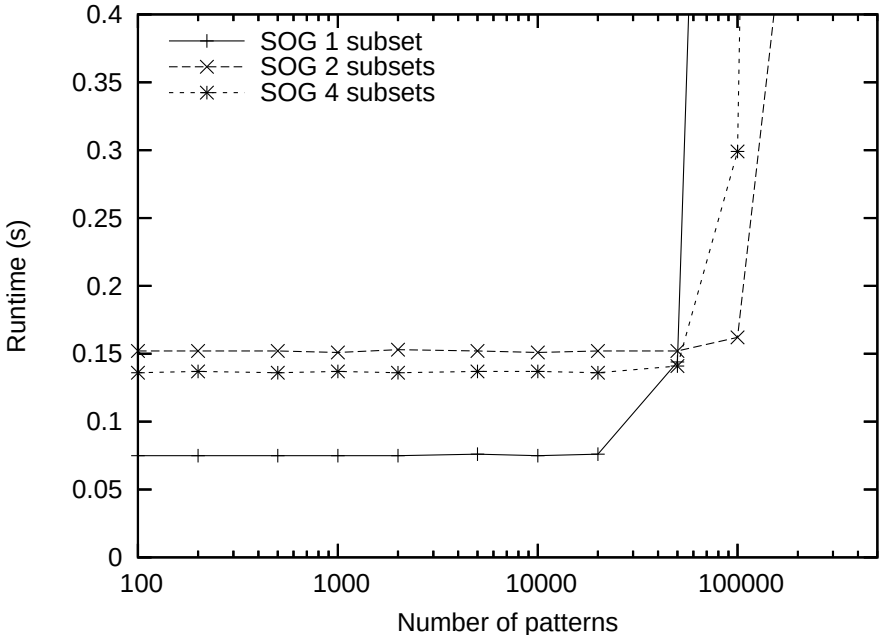


(b) $\sigma = 4$

Figure 5.4: The effect of the parameter q in SOG



(a)



(b)

Figure 5.5: The SOG algorithm. (a) The effect of pattern length. (b) The effect of one, two, and four subsets.

The use of subsets with the SOG algorithm was tested for $m = 8$. We tried versions with one, two, and four subsets, see Figure 5.5(b). The versions using two and four subsets have a better filtering efficiency, and thus their run time remains longer constant when the pattern set size is increased. However, they are again hindered by larger memory requirements. The basic version with one subset needs 64 kB of memory, while the version using two subsets needs 128 kB of memory and the four subsets version 256 kB of memory.

Given r patterns, using four subsets should result in roughly as many false matches as using one subset with $r/4$ patterns because in the version with four subsets only one subset can match at a given position. The results of the tests show that there are a little more matches than that. This is due to the more homogeneous sets produced by the division of patterns.

5.5.2 BG Algorithm

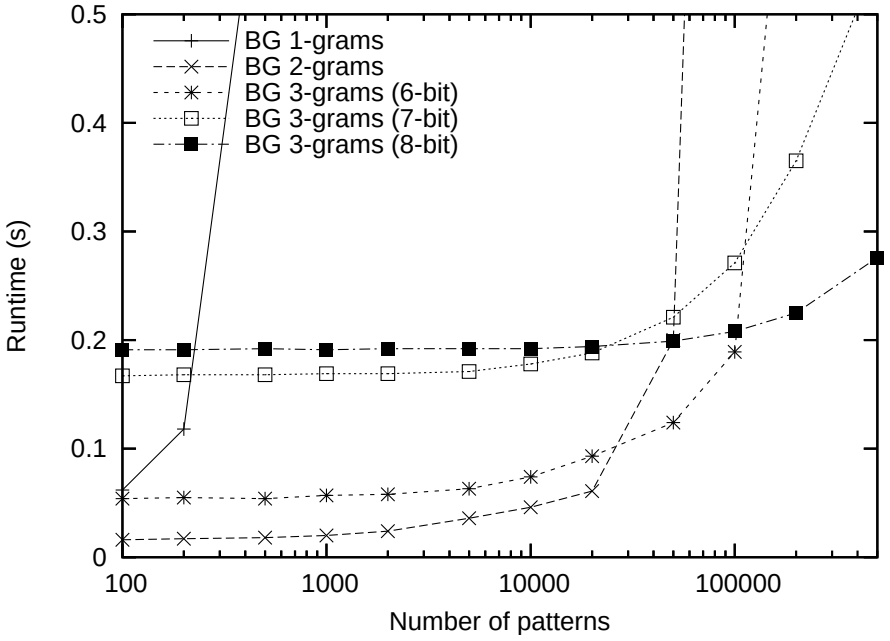
To determine good values of q for the BG algorithm, the same experimental settings as with SOG were used. Figure 5.6 shows the results of these experiments. The analysis predicts that when $\sigma = 256$ and $m = 8$, 1-grams suffice for 100 patterns, 2-grams until 10,000 patterns, and 3-grams should be used after that to keep the verification cost down. Figure 5.6(a) shows that the runtime indeed grows drastically some time after these values. However, there is a slight increase of running time even before that. According to the analysis, if we want to keep the filtering time average optimal, we should use 2-grams when there are less than 10,000 patterns and 3-grams after that. Figure 5.6(a) shows that it is better to use 2-grams until 50,000 patterns, but the runtime starts to increase already earlier. The results for $\sigma = 4$ and $m = 32$ are similar. Comparing with Figure 5.4, we note that the optimal value of q in BG is slightly larger than in SOG.

As with SOG, we also ran some further test with alphabet size 256 using 2-grams. We tested the performance of the BG algorithm for $m = 8$, 16, and 32. Figure 5.7(a) shows the results of these tests. The algorithm is almost as fast in all these cases. The greater memory requirement slows the algorithm down with longer patterns, but on the other hand, longer patterns allow for longer shifts. These two effects seem to balance out each other with smaller pattern sets. When the pattern set grows, the performance degrades faster with shorter patterns because the filtering is less efficient.

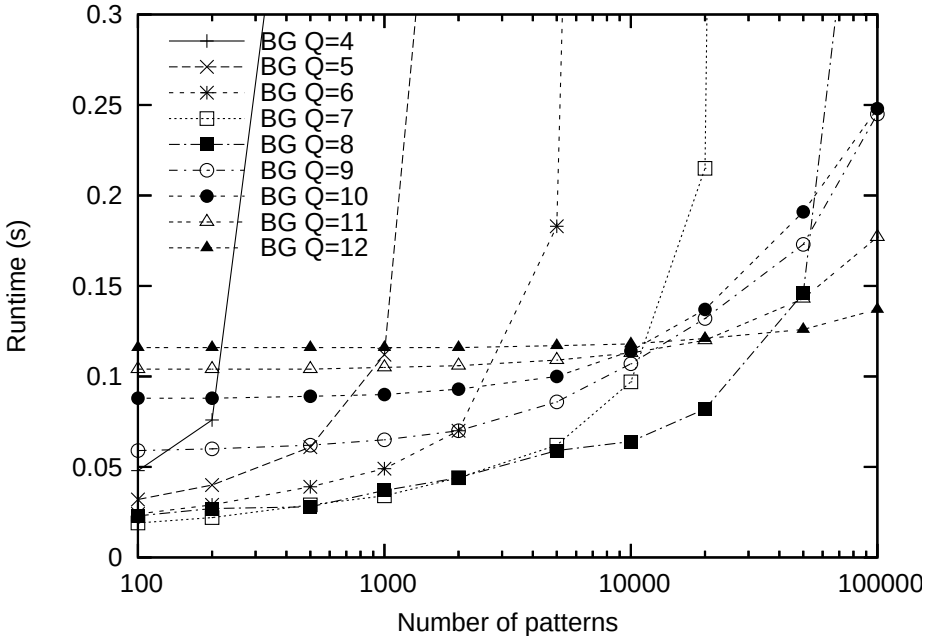
The use of subsets with the BG algorithm was tested for $m = 8$ with one, two, and four subsets, and the results are shown in Figure 5.7(b). The results of these tests are very similar to the ones of the SOG algorithm.

5.5.3 HG Algorithm

We used the same experimental settings as with BG and SOG to determine practical values of q for the HG algorithm. The results of these experiments are shown in Figure 5.8. The 1-gram version is not shown for $\sigma = 256$ because it was not competitive

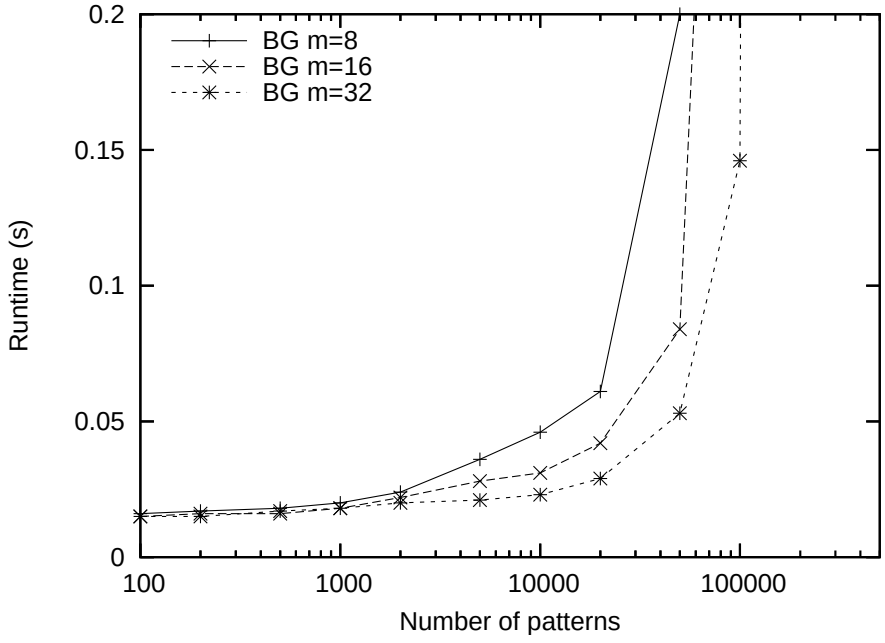


(a) $\sigma = 256$

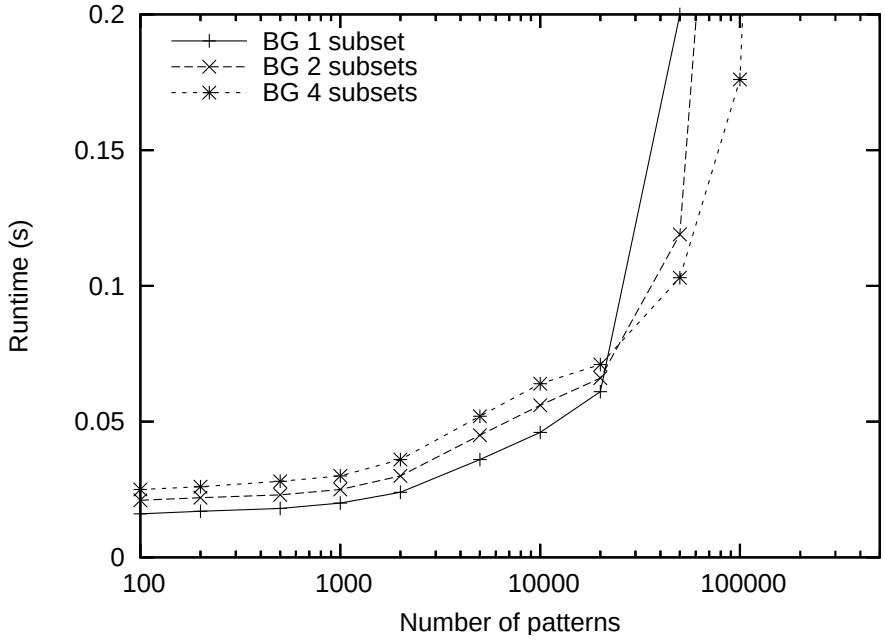


(b) $\sigma = 4$

Figure 5.6: The effect of the parameter q in BG



(a)



(b)

Figure 5.7: The BG algorithm. (a) The effect of pattern length. (b) The effect of one, two, and four subsets.

even when $r = 100$. For HG, the analysis of verification predicts that for $\sigma = 256$ and $m = 8$ when r varies from 100 to 1,000, 2-grams should be used, 3-grams suffice until 100,000 patterns, and 4-grams should be used for larger pattern sets. From Figure 5.8(a), we see that this is a little pessimistic as the runtime increases drastically later than that. For the filtering to be average optimal, the analysis predicts the same values of q as for BG. Again, using 2-grams slightly longer gives better results in practice although HG requires a larger q than BG. The results with the DNA text follow the analysis similarly.

As with SOG and BG, some further tests were run with HG for the alphabet size 256 using 2-grams. Figure 5.9 shows the runtime of the algorithm with different pattern lengths. The times do not include verification of candidates in this case since we implemented the RKBT method only for $m = 8$, 16, and 32. If the verification would be done, the performance of the algorithm would worsen for those set sizes that produce spurious hits. Most of the candidates reported by the HG algorithm are false matches because the probability of finding a real match is very low in our setting.

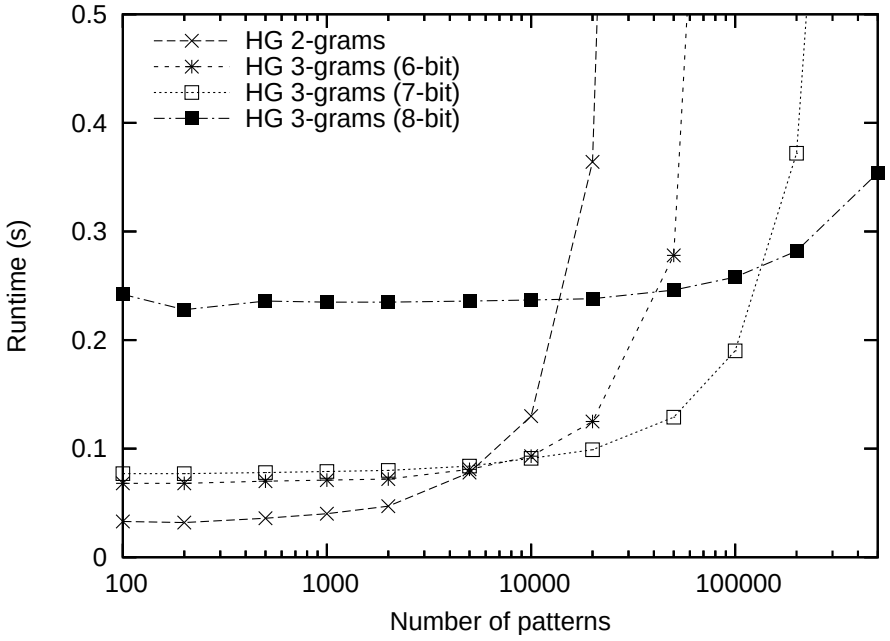
Figure 5.9 shows that when there are less than 10,000 patterns, HG is faster for longer patterns because they allow longer shifts. When the number of false matches grows, the algorithm is faster for shorter patterns because most positions match anyway and the overhead with shorter patterns is smaller.

5.5.4 Comparison of the Algorithms

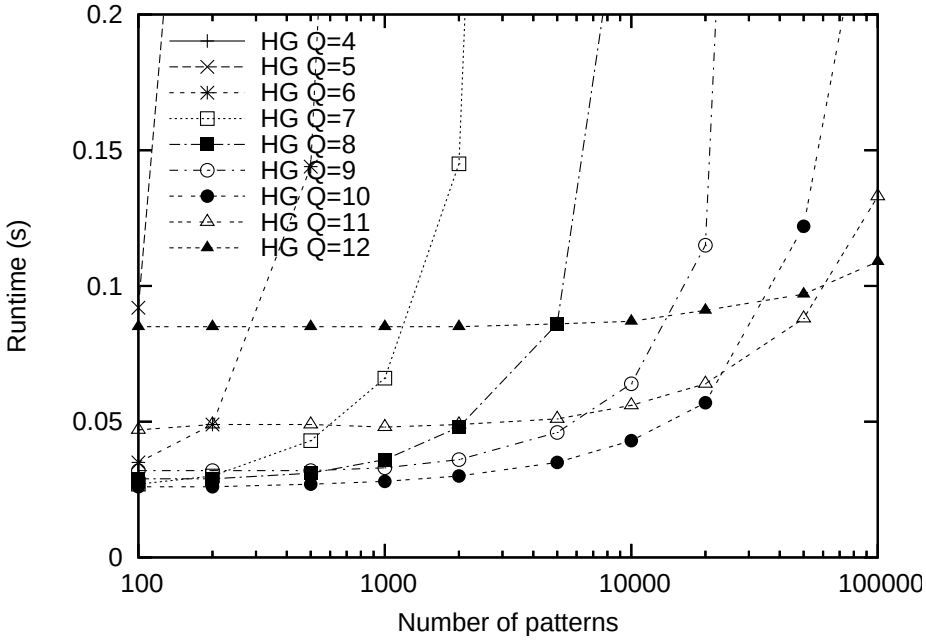
A run time comparison of the algorithms is shown in Figures 5.2 and 5.10(a) based on Table 5.1. These times include verification but exclude preprocessing. The memory usage and the preprocessing times of the algorithms are shown in Table 5.2. These are results from tests with patterns of eight characters, where our algorithms and the Wu-Manber algorithm use the optimal q-value for each pattern set size. Recall that the size of the text is 32 MB.

Figure 5.10(a) shows that our algorithms are considerably faster than the algorithms presented earlier except for the Wu-Manber algorithm, which is only slightly slower than BG, the best of the new algorithms. The HG, BG, and Wu-Manber algorithms are the fastest until 10,000 patterns, while the new algorithms are equally fast between 10,000 and 50,000 patterns. The BG algorithm has the best overall efficiency. With larger pattern sets, the use of subsets with these algorithms would be advantageous.

Table 5.2 shows that the preprocessing phase of our algorithms is fast. Table 5.2 also shows that the memory usage of our algorithms is fairly small, which helps the new algorithms to achieve fast running times because of the hierarchical memory. The memory usage of our filtering techniques is constant for a fixed q. Because our algorithms use RKBT as a subroutine, their numbers cover also all the structures of RKBT including the second hash table. The space increase in Table 5.2 is due to the need to store the patterns for the verification phase and switching from 2-grams to 3-grams. The space for the patterns could be reduced by using clever hash values. For example,



(a) $\sigma = 256$



(b) $\sigma = 4$

Figure 5.8: The effect of the parameter q in HG

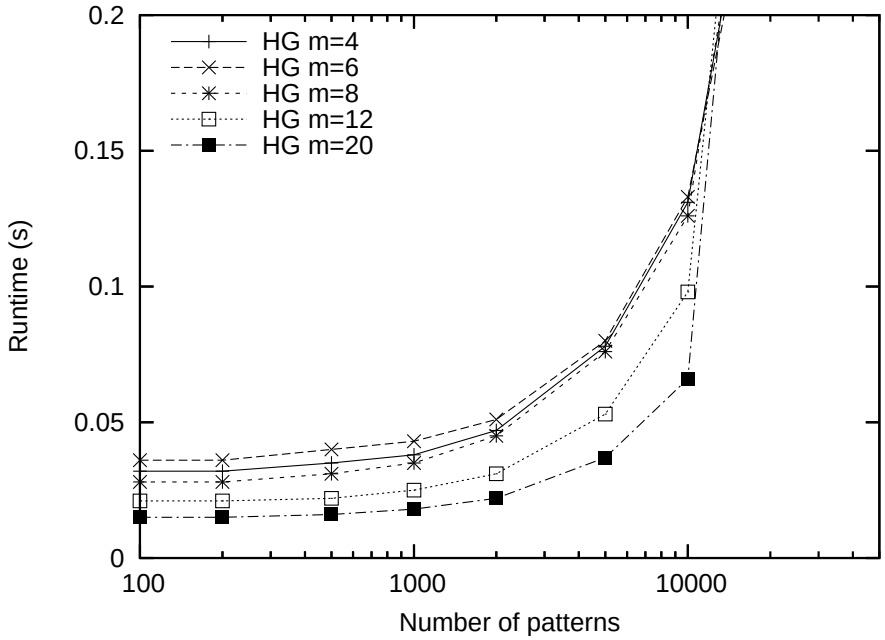
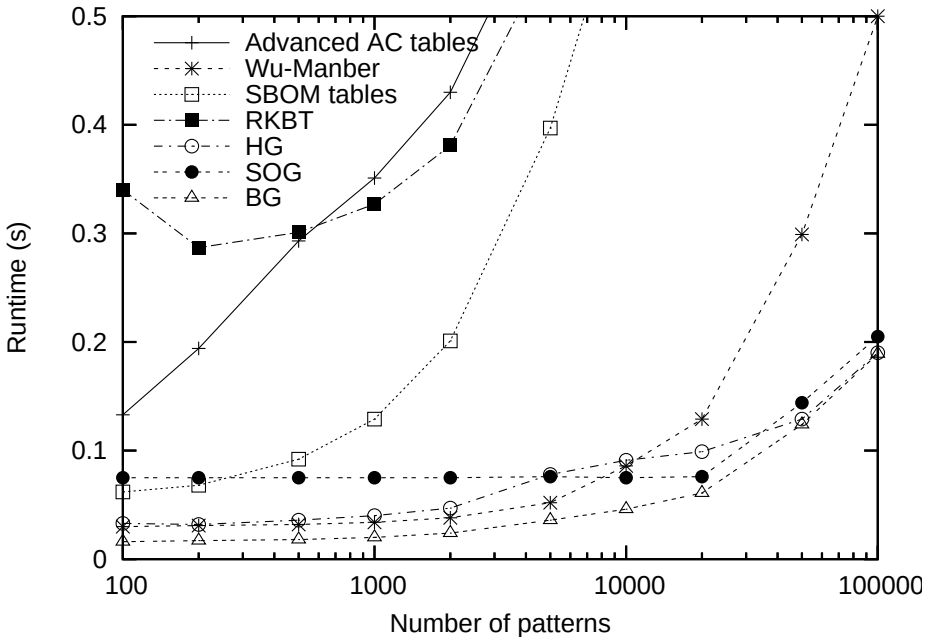


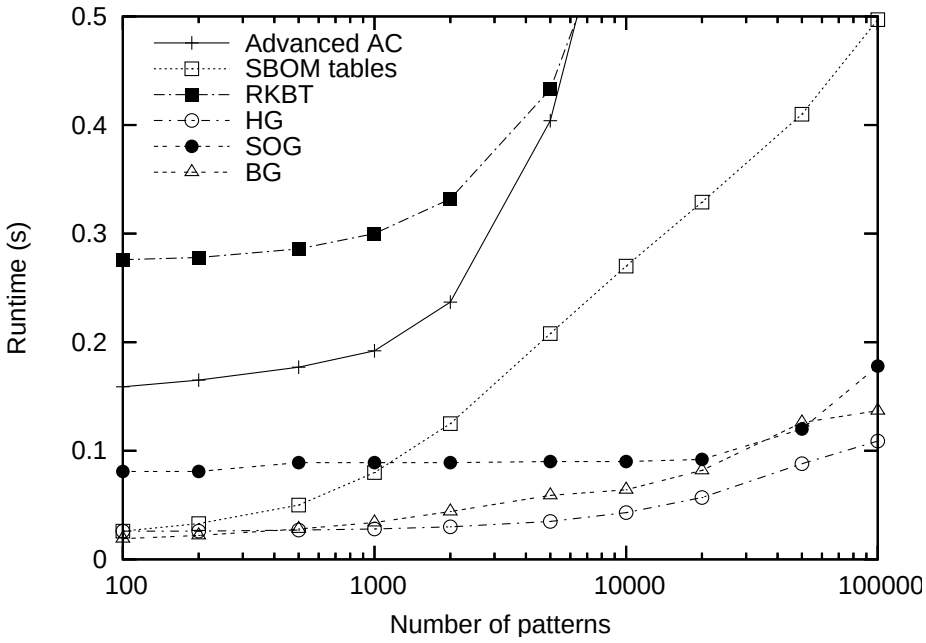
Figure 5.9: Runtimes of the 2-gram version of the HG algorithm for different pattern lengths

Table 5.1: Run times of the algorithms when r varies for $m = 8$ and $\sigma = 256$. AC, Set Horspool, and SBOM algorithms use the table implementation of the trie. Our algorithms use the best observed value of q, and Wu-Manber algorithm uses the best observed sizes of hash tables.

	100	500	1,000	5,000	10,000	50,000	100,000
Advanced AC	0.133	0.293	0.351	0.618	1.053	4.308	6.185
Set Horspool	0.110	0.411	0.493	0.835	1.287	4.360	6.162
Wu-Manber	0.030	0.032	0.034	0.052	0.086	0.299	0.500
SBOM	0.062	0.092	0.129	0.397	0.627	1.163	1.272
RKBT	0.340	0.301	0.327	0.559	0.884	3.412	6.740
HG	0.033	0.036	0.040	0.078	0.091	0.129	0.190
SOG	0.075	0.075	0.075	0.076	0.075	0.144	0.205
BG	0.016	0.018	0.020	0.036	0.046	0.124	0.189



(a) $\sigma = 256, m = 8$



(b) $\sigma = 4, m = 32$

Figure 5.10: Run time comparison of the algorithms. Our algorithms use the best observed value of q, and Wu-Manber algorithm uses the best observed sizes of hash tables.

Table 5.2: Memory usage and preprocessing times of the algorithms for $r = 100$ and 100,000. AC, Set Horspool, and SBOM algorithms use the table implementation of the trie. Our algorithms use the best observed value of q, and Wu-Manber algorithm uses the best observed sizes of hash tables.

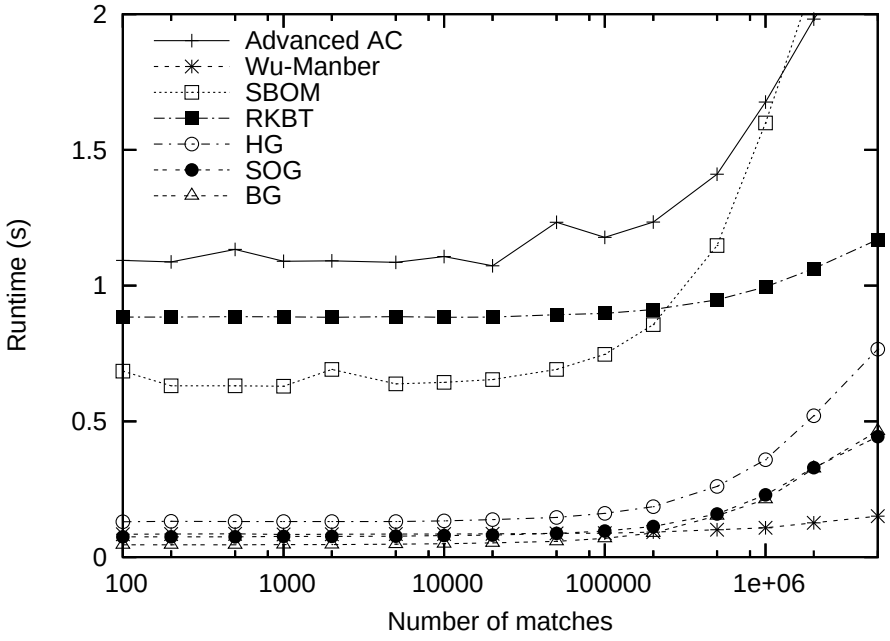
	Memory (kB)		Preprocessing (s)	
Algorithm	$r = 100$	$r = 100,000$	$r = 10,000$	$r = 100,000$
Advanced AC	797	661,496	0.55	12.49
Set Horspool	706	565,743	0.23	1.18
Wu-Manber	326	20,782	0.01	0.08
SBOM	708	571,544	0.39	1.62
RKBT	9	1,180	0.01	0.29
HG	65	2,720	0.01	0.17
SOG	73	1,440	0.01	0.05
BG	73	1,440	0.01	0.05

for $m = 8$ we could store only four characters of each pattern and use a 32-bit hash value such that the other four characters can be obtained from these characters and the hash value.

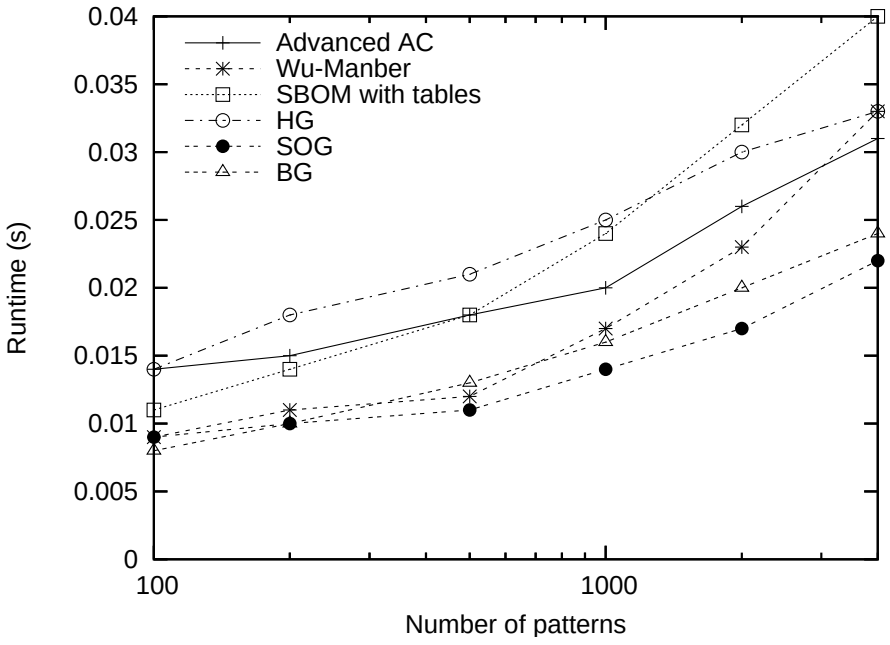
We also run tests on DNA data. The text was a chromosome from the fruit fly genome (22 MB), and we used random patterns of 32 characters. We tried the values 4 through 12 of q in our filtering algorithms. The results using the best observed values for q are shown in Figure 5.10(b). Again the new algorithms are considerably faster than the previous ones.

Our algorithms are filtering methods so they are not designed for searching texts that contain a lot of matches. Nevertheless, we tested the algorithms also in a setting where the text contained matches. Results of these tests show that our algorithms perform surprisingly well also in this setting, see Figure 5.11(a).

To further test our algorithms with a text containing matches, we ran several tests on English text getting somewhat controversial results. We used the King James version of the Bible as a text. First, we used patterns that were formed from at least 8 character long words from the text. Because our algorithms require the patterns to be of equal length, we used 8 character long prefixes of the words. There were 4,216 distinct prefixes. Figure 5.11(b) shows the results of this experiment. The Rabin-Karp method was not competitive in this setting and is thus not shown in the figure. The Set Horspool algorithm, which is also not shown in the figure, was a bit slower than the SBOM algorithm. As the figure shows, Wu-Manber is the fastest of the earlier methods, which confirms with earlier results [80]. However, SOG and BG are equally fast until 1,000 patterns and slightly faster after that.



(a)



(b)

Figure 5.11: (a) Run times of the algorithms when using 10,000 patterns and a text containing a variable amount of matches. (b) Run times of the algorithms when searching English words from the King James version of the Bible.

In the other experiments with English text, we used 8 character long strings randomly chosen from the text. In these tests, the traditional algorithms performed faster than our new ones. The good performance of our algorithms in the first test is probably due to the patterns not containing any space characters, which are very frequent in the text. This allows our algorithms to filter out most of the text positions.

In all these tests, we used pattern sets where all patterns were of equal length. It is worth noting that the new algorithms are not as flexible as for example the Aho-Corasick algorithm when handling patterns of varying length.

5.5.5 Comparison Against the Suffix Array

We also compared BG against the suffix array [67]. We chose the BG algorithm because it was the fastest for random data with alphabet size 256, and it also wins over HG for DNA data if preprocessing time is taken into account. The suffix array implementations are from the *PizzaChili* site[2]. The first implementation, SAu, uses 32-bit integers for suffix array entries, while the second one, SAc, uses $\log n$ bits to represent each entry. We ran this experiment both for DNA data and random data with alphabet size 255.[3] The DNA data for this experiment was also obtained from the *PizzaChili* site. All DNA patterns used in this experiment were of length 32, and the patterns for the experiment with alphabet size 255 were of length 8.

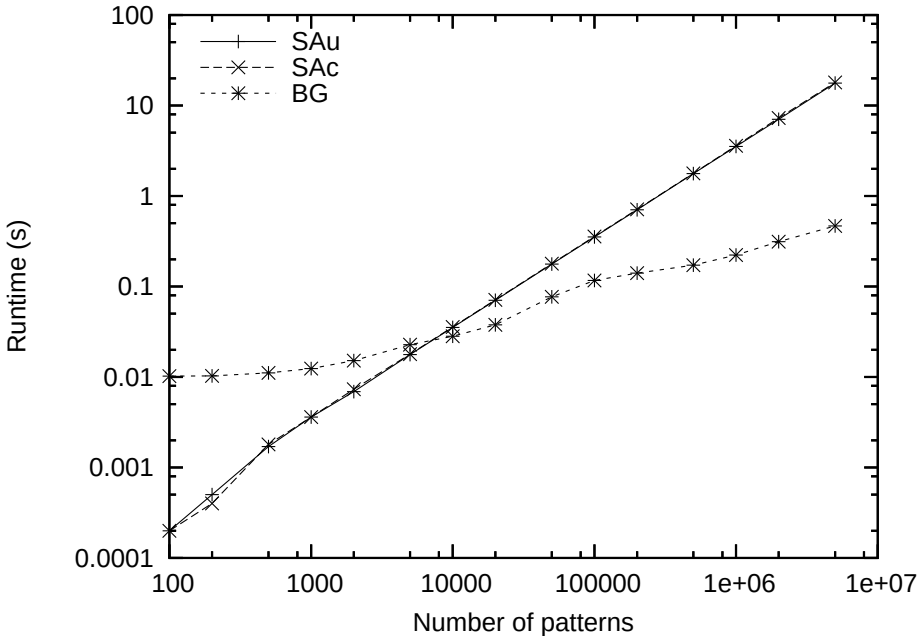
Figures A.1, A.2, and A.3 show the results of these experiments for random data with alphabet size 255, and Figures A.4, A.5, and A.6 show the results for DNA data. Figures A.1 and A.4 show a comparison of search times, Figures A.2 and A.5 a comparison of combined preprocessing and search times, and Figures A.3 and A.6 show a comparison where preprocessing is included in the search time of our algorithms but not in the search time of the suffix array.

As can be seen, BG is superior when comparing the combined preprocessing and search times. If the preprocessing time of the suffix array is excluded, we can see that our approach is faster until some text length which depends on the data and the number of patterns. The more patterns we have, the longer the text must be for the suffix array to be competitive. If the preprocessing time for BG is included, it is beneficial to index somewhat shorter texts than if the preprocessing time is excluded.

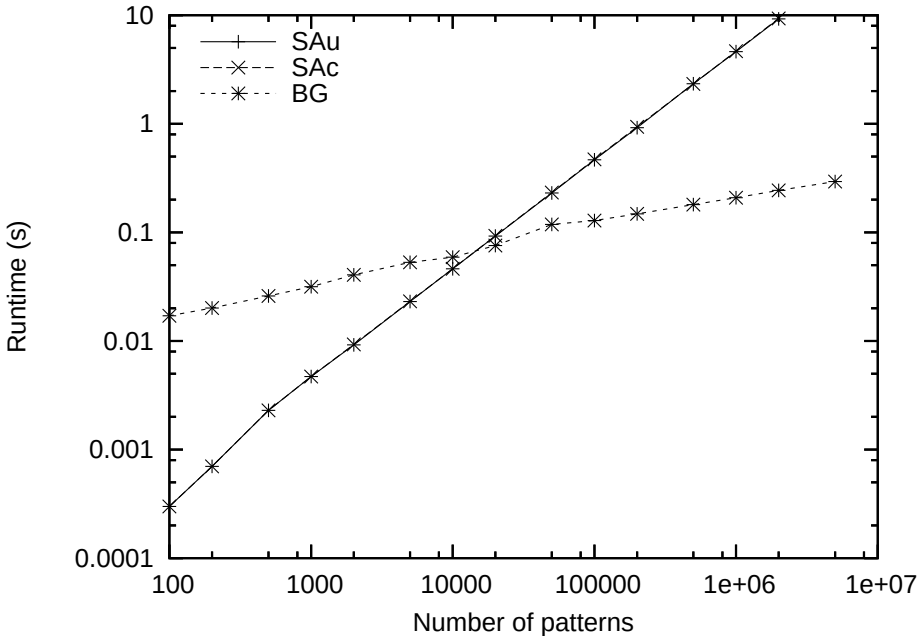
Figure 5.12 further shows a comparison of BG and suffix array for random data with alphabet size 255 and DNA data. For these experiments, we used a text of length 20 MB and varied the number of patterns. The figures show the runtime excluding preprocessing. For this setting, the multiple string matching approach is better if the number of patterns is at least 10,000 for alphabet size 255 or 20,000 for DNA data. Overall, these experiments show that matching multiple patterns simultaneously is a competitive alternative to indexing.

[2] `http://pizzachili.dcc.uchile.cl/`

[3] The suffix array implementations had difficulties with the null character, and thus an alphabet of 256 characters could not be used.



(a) Random data, $\sigma = 255$



(b) DNA data, $\sigma = 4$

Figure 5.12: Runtime comparison of BG and suffix array excluding preprocessing times

Chapter 6

Weighted String Matching

In this chapter, we consider the matching of weighted patterns against an unweighted text. For each position, a weighted pattern assigns a weight to each character of the alphabet, and the weighted pattern matches a string if the score, which is the sum of the weights in the pattern corresponding to the aligned characters in the string, is larger than some given score threshold. The weight of a character can be, for example, the logarithm of the probability of that character occurring at that position, and then the score of a text substring is the logarithm of the probability of that substring matching the pattern.

We adapt some standard string matching algorithms to this problem. We consider two bit-parallel algorithms, shift-add and BNDM. The developed algorithms are similar to the bit parallel algorithms for (δ, γ)-matching [31]. We also consider the enumeration of all strings matching a given weighted pattern and searching for these strings by the multiple string matching algorithms developed in the previous chapter.

6.1 Preliminaries

6.1.1 Definitions

Definition 6.1. *A* weighted pattern *of length m is an $m \times \sigma$ matrix P of integer coefficients $P[i, c]$, which give the weight of the character $c \in \Sigma$ at position i, where $1 \leq i \leq m$.*

We will denote by $P_{i...j}$ a weighted pattern which consist of the weights of the pattern P from position i to position j including positions i and j. If $j < i$, the pattern $P_{i...j}$ has length 0.

Figure 6.1 shows an example of a weighted pattern. Here we will only consider weighted patterns with integer weights. Weighted patterns are obtained from entropy or log odd matrices that have real coefficients, but in practice, these are rounded to integer matrices to allow for more efficient computation.

i	1	2	3	4	5	6	7	8	9	10	11	12
a	7	−6	−5	−10	−8	−10	4	−10	−10	−2	−10	−10
c	−5	−8	−10	14	−10	−8	−10	−10	−10	11	−10	−10
t	6	13	−10	−8	−10	12	−10	−10	−10	−3	−10	9
g	−5	−6	13	−10	14	−1	11	14	14	−10	14	6

Figure 6.1: An example weighted pattern corresponding to the EGR-1 family extracted from TRANSFAC [70]

Definition 6.2. *Given a weighted pattern P of length m and a string $S = s_1 \dots s_m$ of length m drawn from the alphabet Σ, the* score *of the pattern aligned with the string is*

$$\text{score}(P, S) = \sum_{i=1}^{m} P[i, s_i] \ .$$

Problem 6.3. *Given a weighted pattern P of length m, a score threshold α, and an unweighted text $T = t_1 \dots t_n$, the* weighted string matching problem *is to find all such substrings $t_i \dots t_{i+m-1}$ of the text that* $\text{score}(P, t_i \dots t_{i+m-1}) \geq \alpha$.

Given a weighted string matching problem, p-value [27, 97] is a measure that can be used to estimate the statistical significance of the returned substrings.

Definition 6.4. *Given a weighted string matching problem with pattern P and score threshold α, p-*value(P, α) *is the probability that a given background model produces a string S such that* $\text{score}(P, S) \geq \alpha$.

In this work, we assume that the background model is the standard random string model, where each character of the string is chosen independently and uniformly at random. In this case, the p-value can be computed with the following recursion [63]:

$$\begin{aligned} p\text{-value}(P_{1\dots 0}, \alpha) &= \begin{cases} 1 & \text{if } \alpha \leq 0, \\ 0 & \text{otherwise.} \end{cases} \\ p\text{-value}(P_{1\dots i}, \alpha) &= \frac{1}{\sigma} \sum_{c \in \Sigma} p\text{-value}(P_{1\dots i-1}, \alpha - P[i, c]) \end{aligned}$$

6.1.2 Related Work

The brute force algorithm for the weighted string matching problem calculates the score of aligning each substring of the text against the pattern and reports those substrings that yield a score higher than the score threshold. Lately various techniques have been proposed to speed up this scheme. Here we will review those techniques that are relevant to our work. See [84] for a survey on previous work.

Several algorithms use the lookahead technique [110], which provides a way to prune the calculation in a single alignment. For all suffixes of the pattern, there is a maximum score that they can contribute to the overall score. If after matching the prefix of the pattern, the score is not at least the score threshold minus maximum score of the suffix, there cannot be a match at this alignment. By calculating the maximum score for each pattern suffix, the overall computation time can be significantly reduced.

Pizzi et al. [83] have developed an algorithm based on enumerating all matching strings and searching for these with a multi pattern algorithm similar to our enumeration approach. However, they use a different multiple string matching algorithm to search for the enumerated strings, while we use an algorithm tuned for very large pattern sets and low expected number of occurrences.

In Section 6.5, we will compare our algorithms to the algorithm by Liefooghe et al. [63]. Their algorithm uses the lookahead technique, and additionally, it divides the pattern into submatrices and precalculates for all possible strings the score yielded by each submatrix. For example, if we had a pattern of length 12, we could divide it to three submatrices of length four and then precalculate the scores of each submatrix for all the σ^4 possible strings. At matching time, we can then just lookup the scores of each submatrix in a table.

6.1.3 Bit-Parallel Algorithms for Approximate String Matching

We will adapt two bit-parallel approximate string matching algorithms, shift-add [11] and ABNDM [79], to weighted string matching. In this section, we review these algorithms for the k-mismatch problem.

Shift-Add

The shift-add algorithm [11] is an adaptation of the shift-and algorithm for the k-mismatch problem. For each pattern position i from 1 to m, the algorithm has a variable E_i indicating with how many mismatches the suffix of length i of the text read so far matches the pattern prefix of length i. If each of the variables E_i can be represented in b bits, we can concatenate all these variables into a single state vector $E = E_m E_{m-1} \dots E_1$ of length mb. Given a pattern $P = p_1 \dots p_m$, we initialize for each character c in the alphabet a descriptor bit vector $B[c]$, where the bits in the position of E_i are 0^b if c equals p_i and $0^{b-1}1$ otherwise. The vector E (and hence also the variables E_i) can then in the matching phase be all updated at the same time when the next character c is read from the text:

$$E = (E \ll b) + B[c] \ .$$

The algorithm has found a match if $E_m \leq k$.

If the variables E_i count mismatches, the maximum value that they can reach is m, but in the k-mismatch problem, it is enough to be able to represent values in the range

$[0, k+1]$, yielding $b = \lceil \log(k+1) \rceil$. However, we need an additional bit so that the possible carry bits do not interfere with the next variable. With this modification, the update operation of the algorithm becomes:

$$\begin{aligned} E &= (E \ll b) + B[c] \\ of &= (of \ll b) \mid (E \,\&\, (10^{b-1})^m) \\ E &= E \,\&\, (01^{b-1})^m \ . \end{aligned}$$

Here the first line updates the variables E_i, the second one keeps track of those variables E_i that have overflowed, and the last one clears the carry bits. When checking for a match, we now also need to check that the variable E_m has not overflowed, which can be seen from the of vector. The shift-add algorithm for the k-mismatch problem has time complexity $\mathcal{O}\left(n \left\lceil \frac{mb}{w} \right\rceil\right)$, where $b = \lceil \log(k+1) \rceil + 1$ and w is the size of the computer word in bits.

ABNDM

The approximate BNDM (ABNDM) algorithm [79] adapts the BNDM algorithm for approximate matching. Here we describe the version for the k-mismatch problem.

As in the BNDM algorithm, the text is processed in windows of length m, which are read backward. In each window, our goal is to recognize the longest suffix of the window that matches a prefix of the pattern. In order to avoid reading extra characters, we will also need to recognize when the suffix of the window does not match any factor of the pattern, as we can stop processing the window at that point. After a window is processed, we can shift the pattern based on the longest suffix of the window that matches a prefix of the pattern.

To reach this end, the algorithm has a variable E_i for each position of the pattern. If we have read the j last characters of the window, then the variable E_i holds the number of mismatches needed to align the factor $p_i \ldots p_{i+j-1}$ against the j last characters of the window. If E_1 is less than or equal to k, the prefix $p_1 \ldots p_j$ matches the suffix of the window. When we read a new character c, the variables E_i are updated simultaneously as follows:

$$E_i = E_{i+1} + \begin{cases} 0 & \text{if } c = p_i, \\ 1 & \text{otherwise.} \end{cases}$$

If the variables E_i can be represented in b bits, we can concatenate them all into a single bit vector $E = E_1 E_2 \ldots E_m$ of length mb. During preprocessing, we now initialize for each symbol c of the alphabet a descriptor bit vector $B[c]$, where the bits in the position of E_i are 0^b if c equals p_i and $0^{b-1}1$ otherwise. In each window during the matching phase, the vector E is now initialized to $B[c]$, where c is the last character of the window, and the vector is updated as follows when a new character c is read from the window:

$$E = ((E \ll b) \mid (k+1)) + B[c] \ ,$$

where the bitwise or with $k+1$ invalidates those variables that are not valid. If $E_1 \leq k$, then the suffix of the window matches a prefix of the pattern. If all variables $E_i > k$, the suffix of the window does not match any factor of the pattern with at most k mismatches.

Similarly to the shift-add algorithm, it is sufficient that the variables E_i can represent values in the range $[0, k+1]$. This can be achieved by using $b = \lceil \log(k+1) \rceil + 1$ bits, where the last bit is a carry bit, and clearing the carry bits similarly to the shift-add approach. If we now store the distances plus $2^{b-1} - (k+1)$, we can check in constant time if all the distances are greater than k by checking that all the carry bits are set.

6.2 Weighted String Matching with Positive Restricted Weights

The bit-parallel algorithms have problems dealing with negative numbers. Thus, we now define a restricted version of the weighted string matching problem that is more easily solved by the bit-parallel algorithms.

Problem 6.5. *The* weighted string matching problem with positive restricted weights *is a weighted string matching problem, where the weights have the following properties:*

1. $\forall i, 1 \leq i \leq m, \forall c \in \Sigma,\ 0 \leq P[i,c] \leq \alpha$,
2. $\forall i, 1 \leq i \leq m, \exists c \in \Sigma$ *such that* $P[i,c] = 0$,

where P is the weighted pattern of length m and α is the score threshold.

Property 1 is needed for the correct operation of the algorithms, while Property 2 merely serves as a way to lower the score threshold and thus lower the number of bits needed for presenting scores as will be seen later.

In the weighted string matching problem, the weights can be, and in practice often are, negative. The following observation points us to a way to transform any weighted string matching problem to a weighted string matching problem with positive restricted weights. Let P be a weighted pattern of length m, and let P' be a weighted pattern such that for some i, $1 \leq i \leq m$, $P'[i,c] = P[i,c] + h$ for all $c \in \Sigma$ and some constant h, and for all $j \neq i$, $1 \leq j \leq m$, and all $c \in \Sigma$ $P'[j,c] = P[j,c]$. Then the following holds for the scores of P and P' aligned with any string S of length m:

$$\text{score}(P', S) = \text{score}(P, S) + h \ .$$

Therefore, the weighted string matching problem for a text T, pattern P, and score threshold α returns exactly the same alignments as the weighted string matching problem for a text T, pattern P', and score threshold $\alpha' = \alpha + h$.

Now given a weighted pattern matching problem with a score threshold α and a pattern P containing any integer weights, we can transform the problem into an equivalent problem with a score threshold α' and a pattern P' containing only non-negative

weights by adding an appropriate constant h to all weights in the same position and by adjusting the score threshold also by h.

To reduce the score threshold, we further transform the pattern so that in each position at least one of the weights equals zero by adding an appropriate negative constant h to all weights in that position and by adjusting the score threshold also by h. Furthermore, if now any weight is larger than the score threshold, it can be truncated to the score threshold without affecting the returned alignments because the score of an alignment cannot get smaller as more characters are read. The scores of those alignments will, however, be lower. As a result, we have transformed a weighted string matching problem into a weighted string matching problem with positive restricted weights.

6.2.1 Weighted Shift-Add

The adaptation of the shift-add algorithm to weighted string matching with positive restricted weights is quite straightforward. Now instead of counting mismatches, we will be calculating scores so the variables E_i contain the score of the suffix of length i of the text read so far as compared to the prefix of length i of the pattern. For the update operation, the bits corresponding to E_i in the preprocessed descriptor bit vectors $B[c]$ now contain the weight of the character c at position i. The update operation is exactly as in the shift-add algorithm for the k-mismatch problem. If after the update operation the score $E_m \geq \alpha$ or the variable E_m has overflowed, a match is reported.

Property 1 of the weighted string matching problem with positive restricted weights states that all weights are non-negative and thus

$$\text{score}(P_{1\ldots i}, t_j \ldots t_{j+i+1}) \leq \text{score}(P_{1\ldots i+1}, t_j \ldots t_{j+i+2}) \ .$$

Because the score can only increase when reading a new character, we can truncate the score values to α. Property 1 further states that all weights are at most α. Thus if we truncate the score values to α, after the update operation the variables $E_i \leq 2\alpha$ so one carry bit is enough. Therefore, we need to reserve $b = \lceil \log \alpha \rceil + 1$ bits for each variable E_i, and the time complexity of the weighted shift-add algorithm is

$$\mathcal{O}\left(n \left\lceil \frac{m(\lceil \log \alpha \rceil + 1)}{w} \right\rceil\right) \ .$$

In practice, weighted patterns are obtained by rounding log-odd or entropy matrices to integer matrices. Thus, the values of the weights depend on how much precision is preserved by this rounding, and furthermore, practical values of the threshold α depend on the weights. Because of the $\lceil \log \alpha \rceil + 1$ factor in the running time, the weighted shift-add algorithm, and also all the other bit-parallel algorithms for weighted matching presented in this work, are somewhat sensitive to the precision of this rounding unlike other algorithms.

6.2.2 Weighted BNDM

ABNDM can be adapted to the weighted string matching problem with positive restricted weights similarly to the shift-add algorithm. As with the shift-add algorithm, the variables E_i now count scores instead of mismatches, and we initialize the descriptor bit vectors $B[c]$ so that the bits corresponding to a variable E_i contain the weight of the character c in position i.

In weighted BNDM, we need to be able to check if the suffix of the window matches any factor of the pattern. As E_i equals the score of aligning the subpattern $P_{i...i+j-1}$ against the suffix of the window, we now need to be able to tell if this match can be extended to become a full match. For this end, we will have to add the maximum scores of the prefix $P_{1...i-1}$ and the suffix $P_{i+j...m}$ to the variable E_i.

Let $\mathrm{max\_score}(i \ldots j)$ be the maximum score of the subpattern $P_{i...j}$. To include the maximum scores of the pattern suffixes $P_{i+j...m}$ in the scores E_i, we initialize the variables with

$$E_i = smax_i = \min(\mathrm{max\_score}(i+1 \ldots m), \alpha)$$

when we start to process a new window. To include the maximum score of the pattern prefix $P_{1...i-1}$ in the scores E_i, we add

$$pmax_i = \min(\mathrm{max\_score}(1 \ldots i-1), \alpha)$$

to the scores E_i before comparison. If for all of the variables $E_i + pmax_i < \alpha$, then the alignment of the suffix of the window against any factor of the pattern cannot be extended to a full match, and thus we can stop processing this window. When $E_1 \geq \alpha$, we have found a pattern prefix that matches the suffix of the window, and if we have traversed the whole window, a match has been found.

Similarly to the weighted shift-add algorithm, we can truncate the values of the variables E_i to α and use one carry bit to handle the overflow of the variables. Thus we need $b = \lceil \log \alpha \rceil + 1$ bits for each of the variables. If we now store the scores plus $2^{b-1} - \alpha$, we can easily check if the scores have exceeded α by checking the carry bits. This is easily done by redefining

$$smax_i = \min(\mathrm{max\_score}(i+1 \ldots m), \alpha) + 2^{b-1} - \alpha \ .$$

The pseudo code for searching in the weighted BNDM algorithm is shown in Figure 6.2.

6.3 Weighted String Matching with Inverted Weights

The bit-parallel algorithms presented in the previous section need many bits to represent the variables E_i. In typical cases, the p-value is low so that not too many matches are returned. Therefore, the score threshold α tends to be fairly close to the maximum

search ($T = t_1 \ldots t_n$, n, $P = p_1 \ldots p_m$, m)

```
1.  i = m
2.  while (i ≤ n)
3.      last = m
4.      j = 1
5.      E = smax + B[t_i]
6.      while (true)
7.          if ((E & 10^{b-1+b(m-1)}) = 10^{b-1+b(m-1)})
8.              if (j = m)
9.                  report an occurrence starting at i − m + 1
10.                 break
11.             last = m − j
12.         if (((E + pmax) & (10^{b-1})^{m-j}(0^b)^{j-1}) = 0)
13.             break
14.         E = (E ≪ b) + B[t_{i-j}]
15.         j = j + 1
16.     i = i + last
```

Figure 6.2: Searching in weighted BNDM. The handling of carry bits is left out because in practice the scores do not exceed 2α.

score that the pattern can produce when aligned against any string. Thus instead of calculating the score, it could be more efficient to calculate how much lower the score is than the maximum possible score.

We now define the inverted pattern $\bar{P}$. The weight of the character c in the position i in the inverted pattern is the difference between the maximum weight of any character in position i in the original pattern and the weight of the character c in position i in the original pattern:

$$\bar{P}[i, c] = \max_{x \in \Sigma}(P[i, x]) - P[i, c] \ .$$

Similarly, the inverted score threshold $\bar{\alpha}$ is the difference between the maximum score of the pattern and the original threshold:

$$\bar{\alpha} = \text{max\_score}(1 \ldots m) - \alpha \ .$$

In the original weighted string matching problem, we were looking for all such substrings $t_i \ldots t_{i+m-1}$ of the text that the score of that substring aligned with the pattern is at least α. The inverted problem returns the same text substrings if we require that the score of the substring aligned with the pattern is at most $\bar{\alpha}$.

Definition 6.6. *Given an inverted weighted pattern $\bar{P}$ and an inverted score threshold $\bar{\alpha}$, the* inverted weighted string matching problem *is to find all such substrings $t_i \ldots t_{i+m-1}$ that* score$(\bar{P}, t_i \ldots t_{i+m-1}) \leq \bar{\alpha}$.

All weights of the inverted weighted pattern are positive and at least one weight in each position is 0. Therefore, the inverted weighted string matching problem is already a inverted weighted string matching problem with positive restricted weights, and so we can apply bit parallel algorithms to this inverted problem. We can solve such a problem with algorithms which are essentially the same as the bit-parallel algorithms for (δ, γ)-matching [31].

6.3.1 Inverted Weighted Shift-Add

To adapt the weighted shift-add algorithm to the inverted version of the problem, we just need to make one change. If after the update operation, the score $E_m \leq \bar{\alpha}$, we have found a match. In this inverted version of the algorithm, we need to implement overflow handling unlike in the weighted shift-add algorithm, because in practice, the variables E_i do overflow in this case.

6.3.2 Inverted Weighted BNDM

The inverted weighted BNDM algorithm is simpler than the weighted BNDM algorithm. We no longer need to add $smax$ and $pmax$ to the score of the factor of a pattern because the minimum score reached by extending the match of the factor is the score of the factor plus zero, but we still need to initialize the variables to $2^{b-1} - \bar{\alpha} - 1$ when we start processing a window. Thus, line 5 of the pseudo code in Figure 6.2 would change to

$$E = init + B[t_i] \ ,$$

where $init$ is a bit vector, where each of the m fields are set to $2^{b-1} - \bar{\alpha} - 1$. A match on line 7 is now detected if the overflow bit is not set:

$$\text{if } ((E \ \& \ 10^{b-1+b(m-1)}) = 0) \ .$$

We simplify line 12 to

$$\text{if } ((E \ \& \ (10^{b-1})^m) = (10^{b-1})^m)$$

and change line 14 to

$$E = ((E \ll b) + B[t_{i-j}]) \mid 2^{b-1} \ ,$$

which sets the overflow bit for the variables that are not valid. As with the inverted weighted shift-add algorithm, we will now also need to implement overflow handling as the variables E_i do now overflow in practice.

enumerate (P, α)

1. recurse(1, 0)

string S

recurse (i, score)

1. if ($\alpha > \text{score} + \text{max\_score}(i...m)$)
2. return
3. if ($i > m$ and $\text{score} \geq \alpha$)
4. add_string(S)
5. else
6. for each $c \in \Sigma$
7. $s_i = c$
8. recurse($i + 1$, score $+ P[i, c]$)

Figure 6.3: Pseudo code for enumerating all strings that produce a score higher than or equal to the score threshold α

6.4 Enumeration Algorithms

For short patterns, it is possible to enumerate all matching strings, which are the strings that produce a score higher than the score threshold when aligned with the weighted pattern. The enumerated strings can then be searched for with an exact multiple string matching algorithm.

The enumeration of matching strings is done with a recursive algorithm. At recursion level i, we have constructed a string of length $i - 1$ that is a possible prefix of a matching string, and we try to expand that prefix with all characters of the alphabet. This way we have to calculate the score of each prefix only once. The recursion can further be pruned with the lookahead technique. Suppose we have enumerated a prefix of length $i-1$ with score score_i, and the maximum score of a suffix of length $m-i+1$ is $\text{max\_score}(i...m)$. If now the score threshold $\alpha > \text{score}_i + \text{max\_score}(i...m)$, then at this branch of the recursion no matching strings can be found. The pseudo code for enumerating the matching strings is given in Figure 6.3.

The number of enumerated strings is often very large so the algorithms presented in Chapter 5 are well suited to this task. We implemented the method both with the SOG and the BG algorithms.

p-value(P, α) gives the probability of a random string to produce a score equal to or greater than α when aligned with the weighted pattern P. If the background model assumes that all characters are chosen independently and uniformly at random, p-value(P, α) gives the proportion of all possible strings for which the score is at least

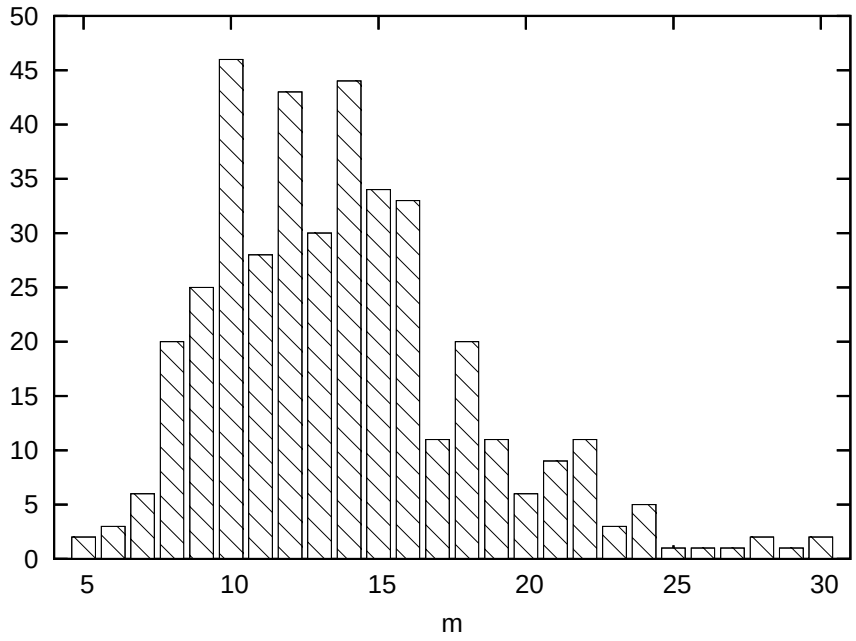


Figure 6.4: The length distribution of patterns in the TRANSFAC database

α. Thus, the expected number of enumerated strings is $\sigma^m \cdot p\text{-value}(P, \alpha)$ because there are σ^m different strings of length m.

In practice, it turned out to be reasonably fast to enumerate matching strings up to pattern length 16. With longer patterns, we enumerated only 16 characters long prefixes of the matching strings, and the algorithm verifies the found matches later.

The enumeration approach is easy to adjust to searching for multiple weighted patterns at once. All we need to do is to enumerate for all of the weighted patterns the strings producing high enough scores and then search for all these enumerated strings.

6.5 Experimental Results

For all experimental testing, we used a computer with a 2.0 GHz AMD Opteron dual-processor and 6 GB of memory. The machine was running the 64-bit version of Linux 2.6.25. The tests were written in C and compiled with the gcc 4.3.0 compiler. The patterns were extracted from the TRANSFAC database [70]. Figure 6.4 shows the length distribution of the patterns. As can be seen, the length of most patterns is between 8 and 22. In particular, there are only a few patterns of length over 22, and thus the results concerning these pattern lengths are only tentative. The text we used was a chromosome from the fruit fly genome (22 MB). The algorithms were run 10 times with each pattern, and the average runtime was calculated. The figures show

average runtimes of patterns of same length. The measured runtimes exclude the time used for preprocessing.

6.5.1 Bit Parallel Algorithms

Figure 6.5 shows a runtime comparison of the bit parallel algorithms, weighted shift-add (wSA), inverted weighted shift-add (iwSA), weighted BNDM (wBNDM), and inverted weighted BNDM (iwBNDM) for two p-values. We see that the runtime of the algorithms increases each time we need more words to represent the state vector. The weighted matching algorithms, wSA and wBNDM, need state vectors of size $\{1, 2, 3, 4, 5\}$ words for pattern lengths $\{5-8, 8-14, 15-21, 19-24, 25-30\}$. Between lengths 19 and 21, some patterns need state vectors of 3 words, while others need 4 words. Similarly for pattern length 8, some patterns need state vectors of 1 word, while others need already 2 words. The number of words needed does not change from the p-value 10^{-3} to the p-value 10^{-5}.

For p-value 10^{-3}, the inverted weighted string matching algorithms, iwSA and iwBNDM, need state vectors of size $\{1, 2, 3, 4, 5\}$ words for pattern lengths $\{5-10, 10-18, 17-24, 22-28, 29-30\}$, respectively. For p-value 10^{-5}, vectors of size $\{1, 2, 3, 4\}$ words are needed for pattern lengths $\{5-14, 11-20, 17-24, 25-30\}$, respectively

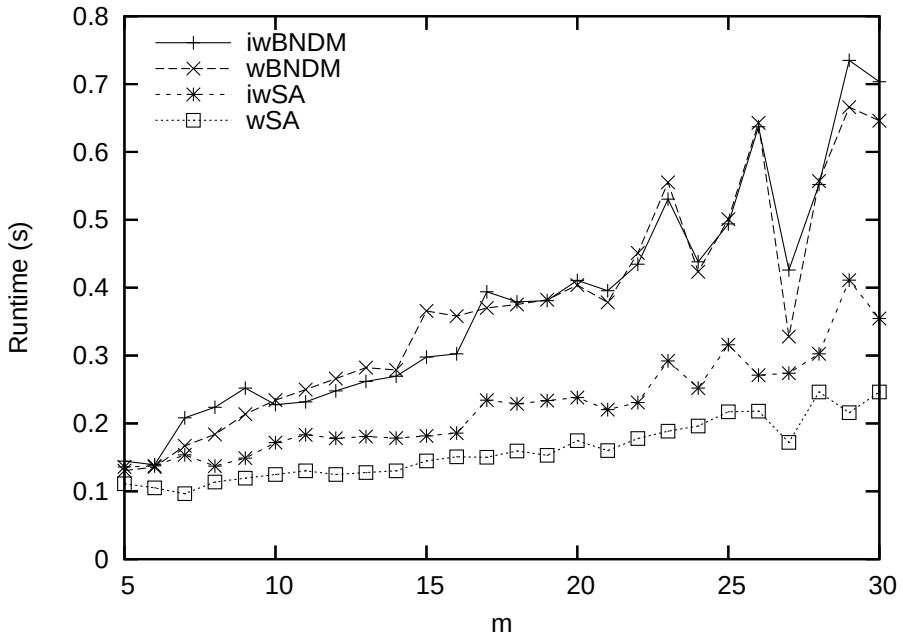
Figure 6.5 shows that the wSA algorithm is faster than the iwSA algorithm. The iwSA algorithm requires less words to represent the state vector, but the need for overflow handling still makes it slower than the wSA algorithm. The runtimes of wBNDM and iwBNDM are much closer. For higher significance levels and longer patterns, iwBNDM clearly takes the lead, but otherwise the differences are not significant.

In almost all situations, the wSA algorithm is the fastest of the bit parallel algorithms. For low significance levels, the wBNDM and iwBNDM algorithms need to read too many characters in each alignment to be competitive, and even for p-value 10^{-5}, they are faster than the wSA algorithm only for pattern lengths from eight to ten.

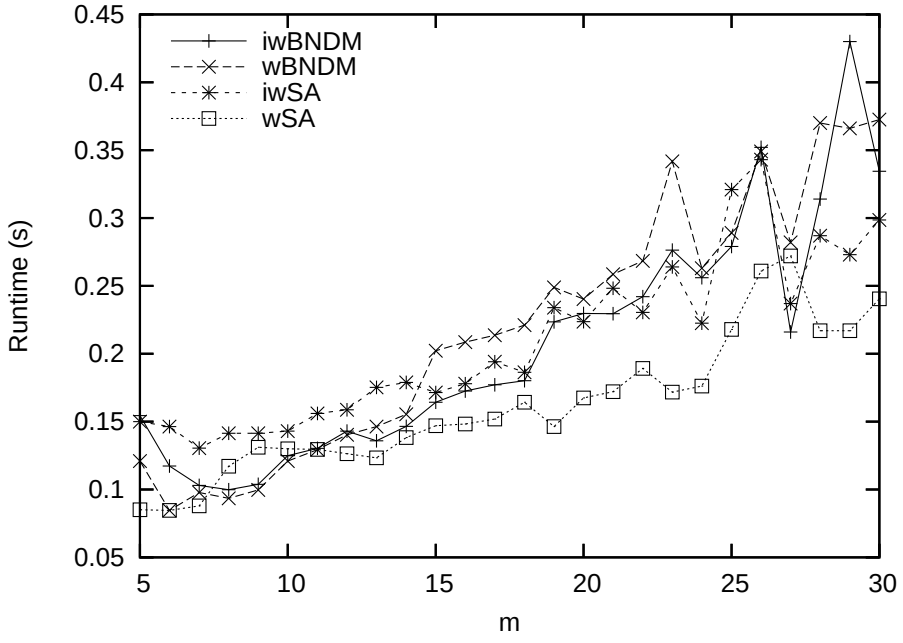
6.5.2 Algorithms for a Single Pattern

Figure 6.6 shows a runtime comparison of the algorithm by Liefooghe, Touzet, and Varré (LTV) [63], weighted shift-add algorithm (wSA), inverted weighted BNDM (iwBNDM), and the enumeration algorithm with BG (eBG) and SOG (eSOG) for two p-values.

For the LTV algorithm, we did not count the optimum length of the submatrices as presented in the original paper by Liefooghe et al. [63] because the optimum length calculation does not take into account cache effects, and these surely have a significant effect on the runtime. Instead, we tried the algorithm with submatrix lengths from 4 to 8 and included the best results in the comparison. With this modification, the method is actually the same as the superalphabet algorithm of Pizzi et al. [83].

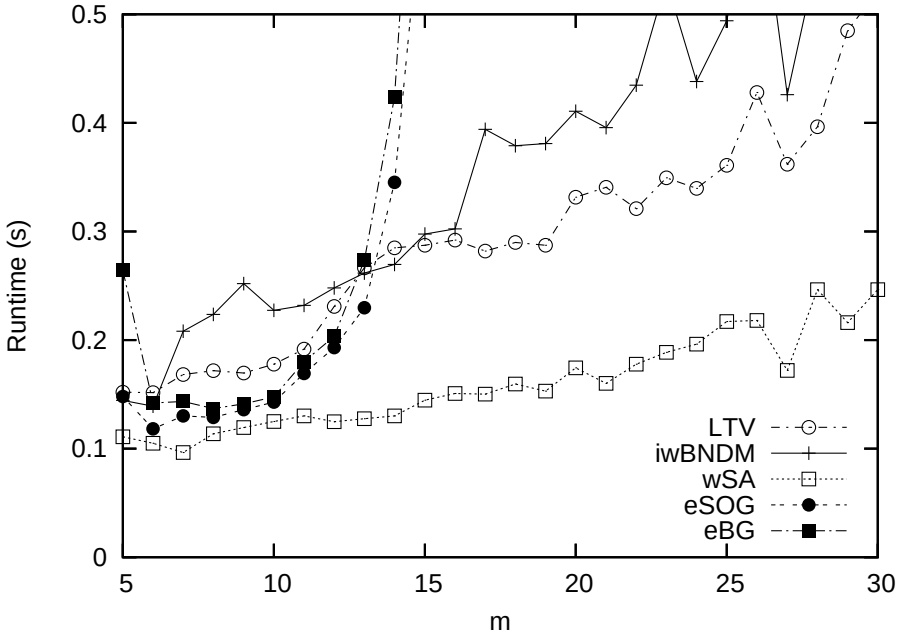


(a)

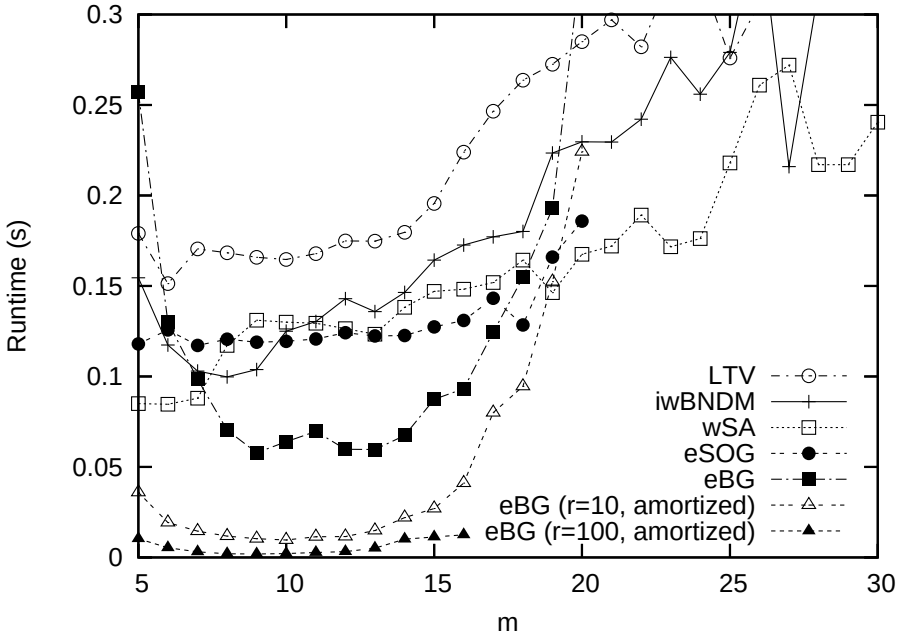


(b)

Figure 6.5: Runtime comparison of the bit parallel algorithms for p-values (a) 10^{-3} and (b) 10^{-5}



(a)



(b)

Figure 6.6: Runtime comparison of different methods for p-values (a) 10^{-3} and (b) 10^{-5}

The optimal value for q in the LTV algorithm is lower for shorter patterns and for higher p-values, but it does not affect the runtime of the algorithm very much until it reaches the value 8 when the tables no longer all fit into the cache. We can see that for the p-value 10^{-3}, the runtime increases slowly until pattern length 11, and for the p-value 10^{-5}, the runtime stays almost constant until pattern length 15. Until that time, it is almost always sufficient to calculate the index of the first precalculated score table corresponding to the first submatrix because the lookahead technique then reports that a match at that position is not possible. When the pattern length increases further, more and more accesses are needed to the second precalculated table until at pattern length 14 for the p-value 10^{-3} and at pattern length 19 for the p-value 10^{-5} at almost every position we need to consult both the first and the second precalculated table.

We ran the enumeration algorithms with several different values of q and chose the value that gives the best runtime. For the p-value 10^{-3} and pattern lengths $\{5 - 7, 8, 9 - 10, 11, 12 - 15\}$, the values $\{4, 5, 6, 7, 8\}$, respectively, gave the best results, and for the p-value 10^{-5} and pattern lengths $\{5 - 10, 11 - 13, 14 - 16, 17 - 20\}$, the values $\{4, 6, 7, 8\}$, respectively, gave the best results when using BG for multiple pattern matching. When using SOG for multiple pattern matching, the optimal value for q was slightly smaller. For the p-value 10^{-3} and pattern lengths $\{5-6, 7-11, 12-14, 15\}$, the values $\{4, 6, 7, 8\}$, respectively, gave the best results, and for the p-value 10^{-5} and pattern lengths $\{5 - 10, 11 - 19, 20\}$, the values $\{4, 6, 8\}$, respectively, gave the best results. We did not run the enumeration algorithms for longer pattern lengths because the number of enumerated patterns grew too large, and already with these pattern lengths, the algorithms started to significantly slow down.

Overall, Figure 6.6 shows that for low significance levels (i.e. high p-values), the weighted shift-add algorithm is the fastest. For higher significance levels (i.e. lower p-values), the weighted shift-add algorithm is the fastest for pattern lengths smaller than 7. The enumeration algorithm with BG is fastest for pattern lengths 8 to 16. For longer patterns, the weighted shift-add algorithm is the fastest at least until pattern length 25. After that the differences between weighted shift-add and LTV are so small that it is hard to say anything conclusive because the TRANSFAC database contained so few long patterns.

The preprocessing of the bit parallel algorithms is very fast taking less than 0.01 s regardless of the pattern length. The preprocessing time for the LTV algorithm ranges from less than 0.01 s to 0.09 s. The preprocessing time of the enumeration algorithms is exponential in the length of the pattern. It stays under 0.01 s until pattern length 13 for the p-value 10^{-3} and until pattern length 16 for the p-value 10^{-5}. For longer patterns, the preprocessing time increases to 0.6 s for the p-value 10^{-3} and pattern length 15 and to 0.4 s for the p-value 10^{-5} and pattern length 20.

6.5.3 Algorithms for Multiple Patterns

We also ran some experiments with the multiple pattern version of the enumeration algorithm using BG. We chose BG for this setting because SOG outperformed BG

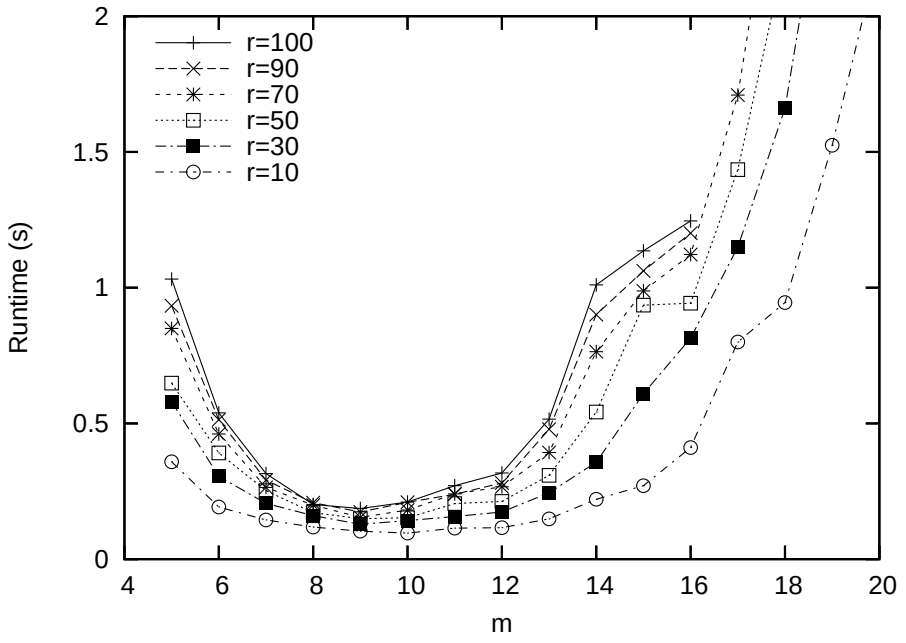
only when the enumeration approach was not the best approach. Because the single pattern algorithm worked well only for high significance levels, we ran the multiple pattern version only for the p-value 10^{-5}. To get reliable results, we needed more patterns of each length than is provided by the TRANSFAC database. To increase the number of patterns for each pattern length, we took prefixes of longer patterns and added these to our pool of patterns until we had a hundred patterns of each length. This worked up to pattern length 16 after which including prefixes of all longer patterns did not bring the number of patterns to one hundred.

Figure 6.7 shows how the runtime of the algorithm behaves as a function of pattern length and pattern set size r. As can be seen, the runtime decreases for all pattern sets as pattern length increases until pattern length 8 because the BG algorithm can make longer shifts. After pattern length 12, the filtering efficiency of the BG algorithm starts to deteriorate, and we need to make more verifications, which increases the runtime. The filtering efficiency could be boosted by increasing the value of parameter q, but this would increase the amount of memory needed so that the structures frequently used by the algorithm no longer fit in the data cache, and this imposes an even larger penalty on the runtime.

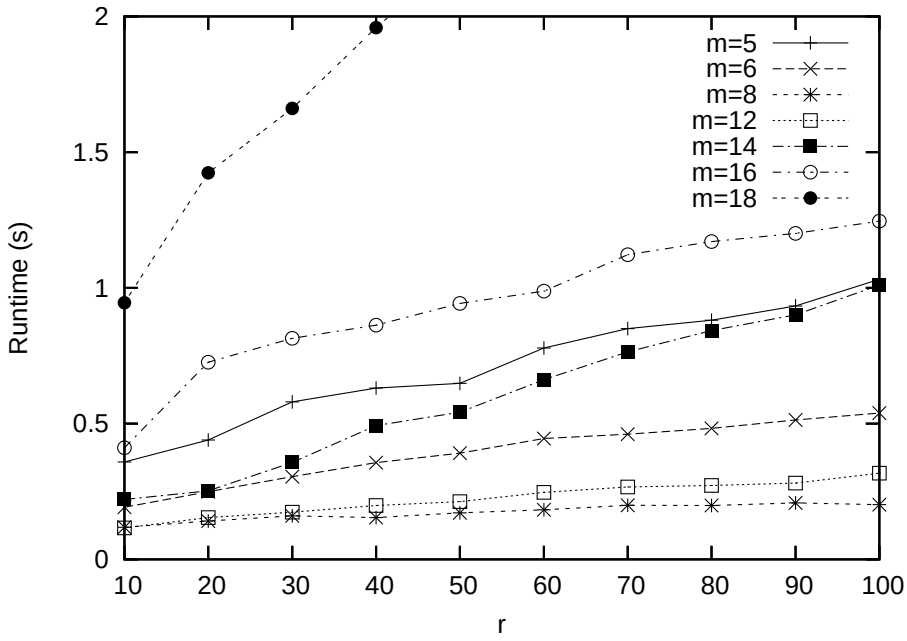
Figure 6.7(b) shows that the runtime increases only slightly when the pattern set size is increased for pattern lengths 6 through 12. For shorter pattern lengths, the performance of the algorithm deteriorates faster because so many positions match at least one of the patterns. For longer patterns, the filtering efficiency is a problem even when searching for a single pattern, and this problem is further emphasized by increasing the pattern set size.

Preprocessing time of the multiple pattern algorithm is less than 0.01 s for all pattern set sizes when the pattern length is at most 11. Figure 6.8 shows the preprocessing times for longer patterns and various pattern set sizes.

The amortized running times (i.e. the running times per pattern) for the multipl pattern enumeration algorithm are shown also in Figure 6.6(b) for pattern set sizes 10 and 100. As can be seen, these times are much lower than the running times of the other algorithms until pattern length 16. After that the runtime starts to increase, and after pattern length 20, it is probably faster to match one pattern at a time using either the shift-add or the LTV algorithm.



(a)



(b)

Figure 6.7: The runtime of the multiple pattern enumeration algorithm as a function of (a) pattern length and (b) pattern set size

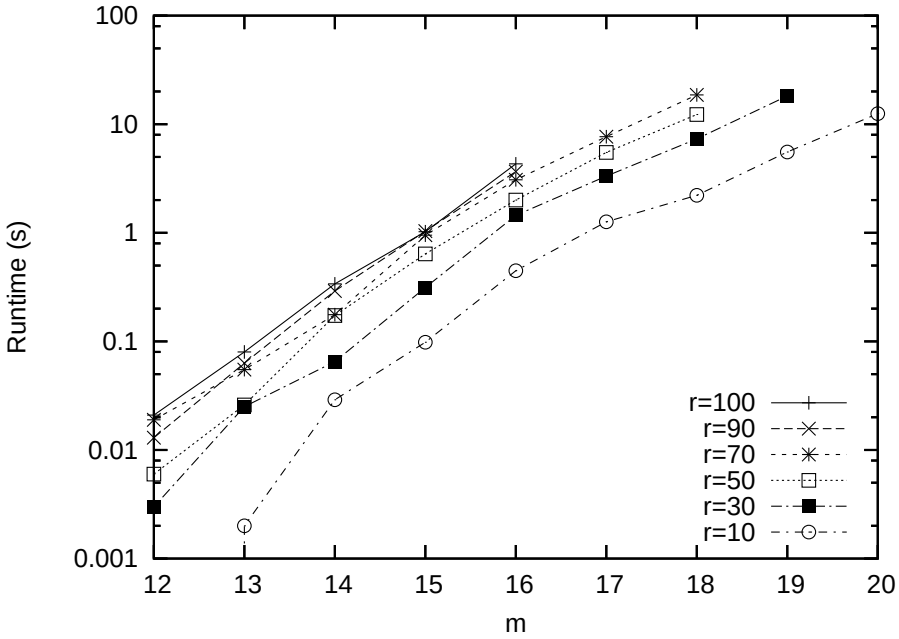


Figure 6.8: Preprocessing times for the multiple pattern enumeration algorithm

Chapter 7

Alphabet Sampling

In the online approach to string matching, the preprocessing of the text is not allowed. Thus these algorithms need to scan the text when searching, and their time cost is of the form $\mathcal{O}(n \cdot f(m))$. The second approach, indexed searching, tries to speed up searching by preprocessing the text and building a data structure that allows searching in $\mathcal{O}(m \cdot g(n) + occ \cdot h(n))$ time, where occ is the number of occurrences of the pattern in the text. Popular solutions to this approach are suffix trees and suffix arrays [67]. The first gives an $\mathcal{O}(m+occ)$ time solution, while the suffix array gives an $\mathcal{O}(m \log n + occ)$ time complexity, which can be improved to $\mathcal{O}(m + occ)$ using extra space [1]. The problem of these approaches is that the space needed is too large for many practical situations (4–20 times the text size). Recently, a lot of effort has been spent to compress these indexes [78], obtaining a significant reduction in space but requiring considerable implementation effort [36].

In this chapter, we explore sampling the text by removing a set of characters from the alphabet. We apply an online algorithm to this sampled text, obtaining an approach in between online searching and indexed searching. We call this kind of structure a *semi-index*. This is a data structure built on top of a text, which permits searching faster than any online algorithm, yet its search complexity is still of the form $\mathcal{O}(n \cdot f(m))$. To be interesting, a semi-index should be easy to implement and require little extra space. Several other semi-indexes exist in the literature, even without using that name. For example, q-gram indexes [76], directly searchable compression formats [71], and other sampling approaches are such semi-indexes.

7.1 Sampled Semi-Index

The main idea of our approach is to choose a subset of the alphabet to be the sampled alphabet and then to build a subsequence of the text by omitting all characters not in the sampled alphabet. At regular intervals, we map the positions of the sampled text to their corresponding positions in the original text. When searching, we build the sampled pattern from the pattern by omitting all characters not in the sampled alphabet and

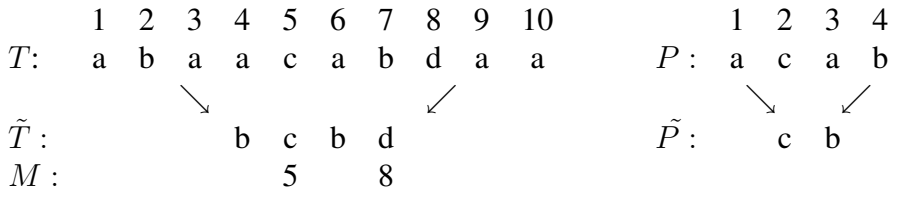


Figure 7.1: Example of preprocessing

then search for this sampled pattern in the sampled text. For each candidate returned by this search, we verify a short range of the original text with the help of the position mapping.

Let $T = t_1 t_2 \dots t_n$ be the text over the alphabet Σ and $\tilde{\Sigma} \subset \Sigma$ the sampled alphabet. The proposed semi-index is composed of the following items:

- Sampled text $\tilde{T}$: Let $\tilde{T} = t_{i_1} t_{i_2} \dots t_{i_{\tilde{n}}}$ be the sequence of the t_i's that belong to the sampled alphabet $\tilde{\Sigma}$. The length of the sampled text is $\tilde{n}$.

- The position mapping M: A table of size $\lfloor \tilde{n}/q \rfloor$, where $M[i]$ maps the character $\tilde{t}_{q \cdot i}$ to its corresponding position in T, so $\tilde{t}_{q \cdot i} = t_{M[i]}$.

Given a pattern $P = p_1 p_2 \dots p_m$, search on this semi-index is carried out as follows. Let $\tilde{P} = p_{j_1} p_{j_2} \dots p_{j_{\tilde{m}}}$ be the subsequence of p_i's that belong to the sampled alphabet $\tilde{\Sigma}$. The length of the sampled pattern is thus $\tilde{m}$. The sampled text $\tilde{T}$ is then searched for $\tilde{P}$, and for every occurrence, the positions to check in the original text are delimited by the position mapping M. If the sampled pattern is found in position i in $\tilde{T}$, the area $t_{M[i/q]+(i \mod q)-j_1+1} \dots t_{M[i/q+1]-(q-i \mod q)-j_1+1}$ is checked for possible startings of real occurrences.

For example, if the text is $T = abaacabdaa$, the sampled text built omitting the a's ($\tilde{\Sigma} = \{b, c, d\}$) is $\tilde{T} = t_2 t_5 t_7 t_8 = bcbd$. If $q = 2$, we map every other position in the sampled text, and then the position mapping M is $\{5, 8\}$. For searching the pattern $acab$, we omit the a's and get $\tilde{P} = p_2 p_4 = cb$. We search for $\tilde{P} = cb$ in $\tilde{T} = bcbd$, finding an occurrence at position 2. The previous mapped position is $M[1] = 5$, so $\tilde{t}_2$ corresponds to t_5, and the next mapped position is $M[2] = 8$, so $\tilde{t}_4$ corresponds to t_8. Because the first sampled character in P is in position 2, we verify the area $4 \dots 5$ in the original text finding the match at position 4. Preprocessing for the text and pattern of the previous example is shown in Figure 7.1.

Because the sampled patterns tend to be quite short, we implemented the search phase with the Boyer-Moore-Horspool algorithm [49], which has been found to be fast in such settings [80]. Figure 7.2 shows the algorithm for this basic method.

search ($\tilde{T} = \tilde{t}_1\tilde{t}_2 \ldots \tilde{t}_{\tilde{n}}$, $\tilde{P} = \tilde{p}_1\tilde{p}_2 \ldots \tilde{p}_{\tilde{m}}$, $T = t_1t_2 \ldots t_n$,
$P = p_1p_2 \ldots p_m$, j_1, q, $M[0 \ldots \tilde{n}/q - 1]$)

1. for ($c \in \Sigma$) $S[c] \leftarrow \tilde{m}$
2. for ($i \leftarrow 1$ to $\tilde{m} - 1$) $S[\tilde{p}_i] \leftarrow \tilde{m} - i$
3. $i \leftarrow 1$
4. while ($i \leq \tilde{n} - \tilde{m} + 1$)
5. $\quad j \leftarrow \tilde{m}$
6. $\quad$ while ($j > 0$ and $\tilde{t}_{i+j-1} = \tilde{p}_j$) $j \leftarrow j - 1$
7. $\quad$ if ($j = 0$)
8. $\quad\quad$ Check for occurrence from $M[i/q] + (i \mod q) - j_1 + 1$
9. $\quad\quad\quad$ to $M[i/q + 1] - (q - i \mod q) - j_1 + 1$
10. $\quad i \leftarrow i + S[\tilde{t}_{i+\tilde{m}-1}]$

Figure 7.2: Searching the sampled text for a sampled pattern with the Boyer-Moore-Horspool algorithm

7.2 Tuning the Semi-Index

Although the above scheme works well for most of the patterns, it is obvious that there are some bad patterns, which would be searched faster in the original text. The average complexity of the Boyer-Moore-Horspool algorithm is

$$\begin{aligned} & n \cdot \left(\frac{1}{m} + \frac{m+1}{2m\sigma} + \mathcal{O}\left(\frac{1}{\sigma^2} \right) \right) \\ = \; & \mathcal{O}\left(n \left(\frac{1}{m} + \frac{1}{\sigma} \right) \right) \\ = \; & \mathcal{O}\left(\frac{n}{\min(m, \sigma)} \right) \end{aligned}$$

assuming a uniform and independent distribution of the characters of the alphabet [9]. If the distribution is not uniform, a better approximation is to replace σ by the *effective alphabet size* $\bar{\sigma}$, which is defined as the inverse of the probability of two random characters matching, i.e. $1/\bar{\sigma} = \sum_{c \in \Sigma} p_c^2$, where p_c is the empirical probability of occurrence of the character c.

We tried several strategies to determine if it would be faster to just search the pattern in the original text. In all cases, we calculated a function $f(\cdot)$ with varying arguments both for the sampled text and the sampled pattern and for the original text and pattern. If the value was better for the original text and pattern, we only search the original text. We tried the following functions:

- $f_1(n, m) = \frac{n}{m}$

- $f_2(n, m, \bar{\sigma}) = \frac{n}{\min(m, \bar{\sigma})}$
- $f_3(n, m, \bar{\sigma}) = n \cdot \left(\frac{1}{m} + \frac{1}{\bar{\sigma}}\right)$
- Based on the empirical probabilities p_c of characters in the text, we calculated the expected shift length for the given pattern

  $$\bar{s} = \sum_{c \in \Sigma} p_c \cdot S[c] \ ,$$

  where $S[c]$ is the bad character function. The compared function is then $f_4(n, \bar{s}) = n/\bar{s}$.
- $f_5(n, m, \bar{\sigma}) = n \cdot \left(\frac{1}{m} + \frac{m+1}{2m\bar{\sigma}}\right)$.

7.3 Optimal Sampling

A question arises from the previous description of our sampling method: How to form the sampled alphabet $\tilde{\Sigma}$? We will first analyze how the average running time of the Boyer-Moore-Horspool algorithm changes when we sample the text, and then based on this, we will develop a method to find the optimal sampled alphabet. Throughout this section, we assume that the characters are independent, and we analyze the approach for a general pattern not known when preprocessing the text.

Let us define

$$\begin{aligned} b_A &= \sum_{c \in A} p_c \\ a_A &= \sum_{c \in A} p_c^2 \ , \end{aligned}$$

where $A \subset \Sigma$. Now the length of the sampled text will be $b_{\tilde{\Sigma}} n$ and the average length of the sampled pattern $b_{\tilde{\Sigma}} m$ if we assume it distributes similarly to the text. The probability of two random characters matching in the sampled text is now

$$\sum_{c \in \tilde{\Sigma}} \left(\frac{p_c}{\sum_{x \in \tilde{\Sigma}} p_x} \right)^2 = \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}^2} \ .$$

Given the average complexity of the Boyer-Moore-Horspool algorithm, $\mathcal{O}(n(1/m + 1/\bar{\sigma}))$, the average search cost in the sampled text is

$$\begin{aligned} & \mathcal{O}\left(b_{\tilde{\Sigma}} n \left(\frac{1}{b_{\tilde{\Sigma}} m} + \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}^2} \right) \right) \\ = \ & \mathcal{O}\left(n \left(\frac{1}{m} + \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} \right) \right) \ . \end{aligned}$$

When considering the verification cost, we assume for simplicity that the mapping M contains the position of each sampled character in the original text, i.e. $q = 1$. For a larger q, the verification cost would increase because the area that we need to verify increases for each triggered verification.

The probability that the sampled pattern is of length i is

$$\binom{m}{i} b_{\tilde{\Sigma}}^{i} (1 - b_{\tilde{\Sigma}})^{m-i} \ .$$

A position in the sampled text triggers a verification if all the characters of the sampled pattern match the substring of the sampled text starting at that position. If the length of the sampled pattern is i, then the probability for this event is

$$\left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}^2} \right)^{i} \ .$$

Hence, the probability that a position has to be verified is

$$\begin{aligned} p_{\text{ver}} &= \sum_{i=0}^{m} \binom{m}{i} b_{\tilde{\Sigma}}^{i} (1 - b_{\tilde{\Sigma}})^{m-i} \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}^2} \right)^{i} \\ &= \sum_{i=0}^{m} \binom{m}{i} \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} \right)^{i} (1 - b_{\tilde{\Sigma}})^{m-i} \\ &= \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}} \right)^{m} \ . \end{aligned}$$

If we assume that each verification costs $\mathcal{O}(m)$, then the cost of verification is

$$\begin{aligned} & n \cdot p_{\text{ver}} \cdot \mathcal{O}(m) \\ = \ & n \cdot \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}} \right)^{m} \cdot \mathcal{O}(m) \ . \end{aligned}$$

The total cost of searching in our scheme is thus

$$\mathcal{O}\left(n \cdot \left(\frac{1}{m} + \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}} \right)^{m} \cdot m \right) \right) \ ,$$

and hence the optimal sampled alphabet $\tilde{\Sigma}$ minimizes the cost per text character

$$E(\tilde{\Sigma}) = \frac{1}{m} + \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}} \right)^{m} \cdot m \ ,$$

which can be divided into the search cost in the sampled text

$$E_{\text{search}}(\tilde{\Sigma}) = \frac{1}{m} + \frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}}$$

and the verification cost

$$E_{\text{ver}}(\tilde{\Sigma}) = \left(\frac{a_{\tilde{\Sigma}}}{b_{\tilde{\Sigma}}} + 1 - b_{\tilde{\Sigma}} \right)^m \cdot m \ .$$

Hence the verification cost always increases when a character is removed from the alphabet, and so the search cost in the sampled text must decrease for the combined cost to decrease. If $R = \Sigma \backslash \tilde{\Sigma}$ is the set of removed characters, the function

$$h_R(p) = \frac{1}{m} + \frac{a_\Sigma - a_R - p^2}{1 - b_R - p}$$

gives the search cost in the sampled text, per text character, if an additional character with probability p is removed. The derivative of $h_R(p)$ is

$$\begin{aligned} h'_R(p) &= \frac{-2p\,(1 - b_R - p) + (a_\Sigma - a_R - p^2)}{(1 - b_R - p)^2} \\ &= \frac{p^2 - 2p\,(1 - b_R) + (a_\Sigma - a_R)}{(1 - b_R - p)^2} \\ &= \frac{(1 - b_R - p)^2 - (1 - b_R)^2 + (a_\Sigma - a_R)}{(1 - b_R - p)^2} \\ &= 1 - \frac{(1 - b_R)^2 - (a_\Sigma - a_R)}{(1 - b_R - p)^2} \ . \end{aligned}$$

We then solve the zeroes of the derivative:

$$\begin{aligned} h'_R(p) &= 0 \\ \frac{(1 - b_R)^2 - (a_\Sigma - a_R)}{(1 - b_R - p)^2} &= 1 \\ p^2 - 2(1 - b_R)p + (a_\Sigma - a_R) &= 0 \\ p &= (1 - b_R) \pm \sqrt{(1 - b_R)^2 - (a_\Sigma - a_R)} \ . \end{aligned}$$

Of these only

$$p_z = (1 - b_R) - \sqrt{(1 - b_R)^2 - (a_\Sigma - a_R)}$$

is in the interval $[0, 1 - b_R]$. We can see that the function $h_R(p)$ is increasing until p_z and decreasing after that. Solving the equation

$$\begin{aligned} h_R(p_R) &= h_R(0),\ p_R \neq 0 \\ \frac{1}{m} + \frac{a_\Sigma - a_R - p_R^2}{1 - b_R - p_R} &= \frac{1}{m} + \frac{a_\Sigma - a_R}{1 - b_R} \ , \end{aligned}$$

we get

$$p_R = \frac{a_\Sigma - a_R}{1 - b_R} \ .$$

So removing a single additional character decreases the search cost in the sampled text only if the probability of occurrence for that character is larger than p_R. Otherwise, both the search cost in the sampled text and the verification cost will increase, and thus removing the character is not beneficial.

Suppose now that we have already fixed whether we are going to keep or remove each character with probability of occurrence higher than p_c, and now we need to decide if we should remove the character c. If $p_c > p_R$, we will need to explore both options as removing the character will decrease search cost in the sampled text and increase verification cost. However, if $p_c < p_R$, we know that if we added only c to R, the searching time in the sampled text would also increase, and therefore we should not remove c. But could it be beneficial to remove c together with a set of other characters with probabilities of occurrence less than p_R? In fact it cannot be. Suppose that we remove a character c with probability $p_c < p_R$. Now the new removed set will be $R' = R \cup \{c\}$, and so we get $a_{R'} = a_R + p_c^2$ and $b_{R'} = b_R + p_c$. Now the new critical probability will be

$$p_{R'} = \frac{a_\Sigma - a_{R'}}{1 - b_{R'}} = \frac{a_\Sigma - a_R - p_c^2}{1 - b_R - p_c} \ .$$

We know that

$$h_R(p_c) > h_R(p_R) = h_R(0)$$

because $p_c < p_R$. Therefore,

$$\begin{aligned} h_R(p_c) &> h_R(0) \\ \frac{1}{m} + \frac{a_\Sigma - a_R - p_c^2}{1 - b_R - p_c} &> \frac{1}{m} + \frac{a_\Sigma - a_R}{1 - b_R} \ , \end{aligned}$$

and so

$$p_{R'} = \frac{a_\Sigma - a_R - p_c^2}{1 - b_R - p_c} > \frac{a_\Sigma - a_R}{1 - b_R} = p_R \ .$$

Thus even now it is not good to remove a character with probability less than the critical value p_R for the previous set, and this will again hold if another character with a small probability is removed. Therefore, we do not need to consider removing characters with probabilities less than p_R. Note, however, that removing a character with a higher probability will decrease the critical probability p_R, and after this, it can be beneficial to remove a previously unbeneficial character. In fact, if the sampled alphabet contains two characters with different probabilities of occurrence, the probability of occurrence for the most frequent character in the sampled alphabet is always larger than p_R. Thus, it is always beneficial for searching in the sampled text to remove the most frequent character.

The above can be applied to prune the exhaustive search for the optimal set of removed characters. First, we sort the characters of the alphabet in the decreasing order of frequency. We then figure out if it is beneficial for searching in the sampled text to remove the most frequent character not considered yet. If it is, we try both

$R_{opt} = \{\}$
sort characters of Σ in descending order of frequency
find_opt(1, {})
return R_{opt}

find_opt (i, R)

```
1.  if (i = σ + 1)
2.      if (E(Σ\R) < E(Σ\R_opt))
3.          R_opt = R
4.  else
5.      p_R = (a_Σ − a_R) / (1 − b_R)
6.      if (p_i > p_R)
7.          find_opt(i + 1, R ∪ {i})
8.          find_opt(i + 1, R)
9.      else
10.         find_opt(σ + 1, R)
```

Figure 7.3: Pseudo code for searching for the optimal set of removed characters

removing and not removing that character and proceed recursively for both cases. If it is not, we prune the search here because none of the remaining characters should be removed. Figure 7.3 gives the pseudo code.

In practice when using this pruning technique, the number of examined sets drops drastically as compared to the exhaustive search, although the worst case is still exponential. For example, the number of examined sets drops from 2^{61} to 2,810 when considering the King James Bible as the text.

In our experiments, the optimal set of removed characters always contained the most frequent characters up to some limit depending on the length of the pattern, as shown in Table 7.1. Therefore, a simpler heuristic is to remove the k most frequent characters for varying k and choose the set that predicts the best overall time. However, if the verification cost is very high for some reason (e.g. going to disk to retrieve the text, or uncompressing part of it), it is possible that the optimal set of removed characters is not a set of most frequent characters.

7.4 Experimental Results

To determine the sampled alphabet, we ran the exact algorithm of Section 7.3 for different pattern lengths to choose the sampled alphabet that produces the smallest estimated cost $E(\tilde{\Sigma})$. For all pattern lengths, the algorithm recommended removing a set of most frequent characters. To see how well these results correspond to practice,

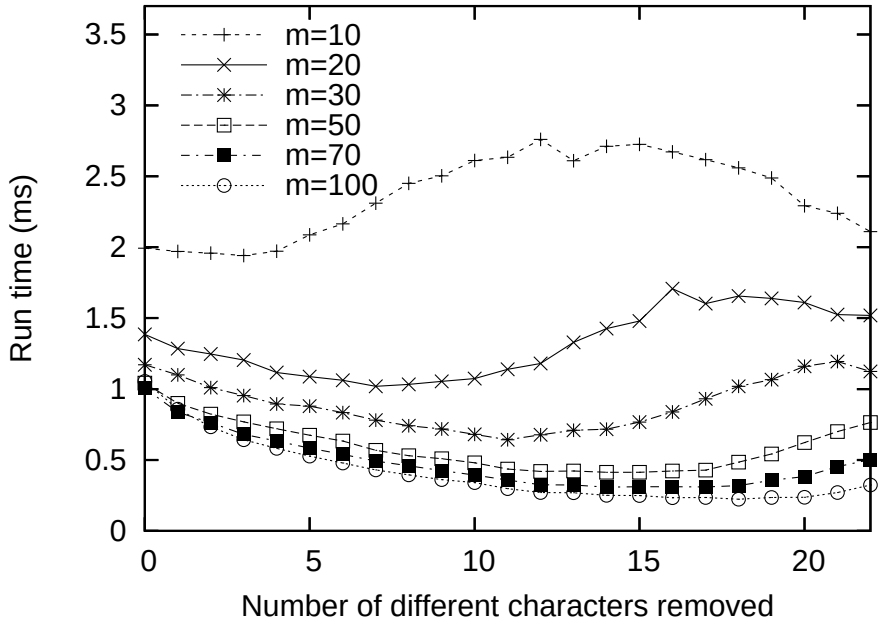
Table 7.1: Predicted and observed optimal number of removed characters for the King James Bible. The predicted optima are computed with the algorithm suggested by the analysis, which in our experiments always returned a set of most frequent characters.

m	10	20	30	40	50	60	70	80	90	100
Predicted by analysis	3	7	9	11	12	13	14	15	16	16
Observed optimum	3	7	11	13	14	15	17	17	16	18

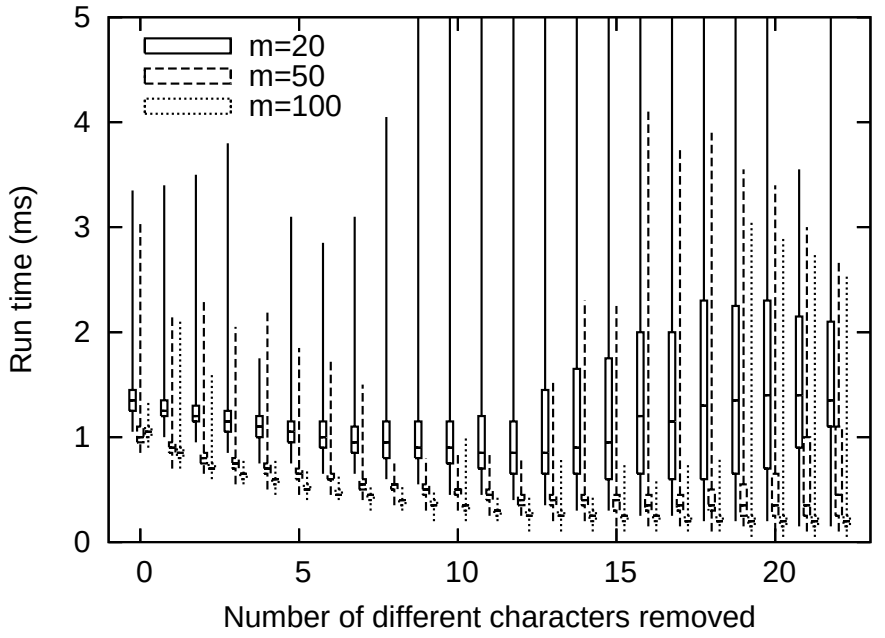
we tested the semi-index approach by removing the k most frequent characters from the text for varying k. We used a 2 MB prefix of the King James Bible as the text, and the patterns are random substrings of the text. For each pattern length, 500 patterns were generated, and the reported running times are averages over 200 runs with each of the patterns. The most frequent characters in the decreasing order of frequency were "␣ethaonsirdlfum,wycgbp", where ␣ is the space character. The tests were run on a 1.0 GHz AMD Athlon dual core processor with 2 GB of memory, 64 kB L1 cache, and 512 kB L2 cache, running Linux 2.6.23. The code is in C and compiled with `gcc` using `-O3` optimization.

Figure 7.4 shows the results of these experiments with the basic method mapping every 64:th sampled character to its position in the original text. If we make the mapping sparser, the running time will start to increase a little earlier, but the effect is quite mild. The results for zero removed characters correspond to the original Boyer-Moore-Horspool algorithm. We see that the semi-index is up to 5 times faster, especially when the patterns are long. We also see that for each pattern length, there is an optimal number of characters to remove. A comparison of these optima and those given by the analysis is shown in Table 7.1. We see that the analysis gives reasonably good results although it recommends removing too few characters with long patterns because we estimated the verification time quite pessimistically. When more characters are removed, it is unlikely that we would need to read m characters for each verified position.

Figures 7.5, 7.6, 7.7, 7.8, and 7.9 show the results for the tuned versions of the sampled semi-index presented in Section 7.2. In these methods, we search the original text if it looks like that will be faster than searching the sampled text. Also in all these tests, every 64:th sampled character is mapped to its position in the original text. Figure 7.5 shows that using the function $f_1(n, m) = n/m$ yields good predictions for short patterns but longer ones are affected adversely. When using the functions $f_3(n, m, \bar{\sigma}) = n \cdot (1/m + 1/\bar{\sigma})$, $f_4(n, \bar{s}) = n/\bar{s}$, and $f_5(n, m, \bar{\sigma}) = n \cdot (1/m + (m + 1)/(2m\bar{\sigma}))$ shown in Figures 7.7, 7.8, and 7.9, respectively, the prediction works well for small number of removed characters, but then the runtime suddenly increases to much more than the runtime of the plain Boyer-Moore-Horspool algorithm. Figure 7.6 shows that using the function $f_2(n, m, \bar{\sigma}) = n/\min(m, \bar{\sigma})$ suffers from the same phenomenon, but it is much milder, making this the best tuned method.



(a) Mean



(b) Distribution

Figure 7.4: The running time for various pattern lengths for the basic method. The top figure shows the mean running time; the bottom figure shows the median, minimum, maximum, and 25% and 75% quartiles.

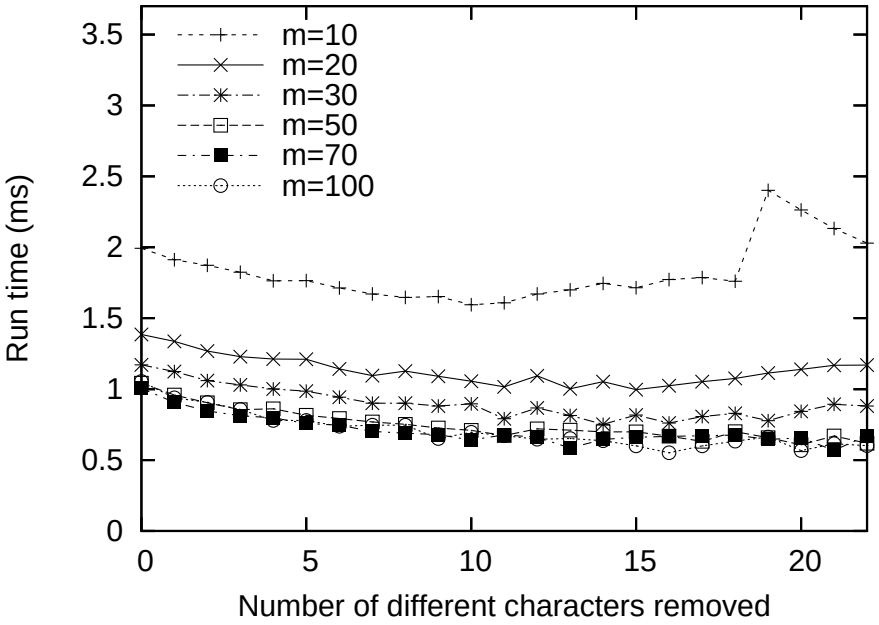


Figure 7.5: Runtime for the tuned version of the sampled semi-index using $f_1(n, m) = n/m$

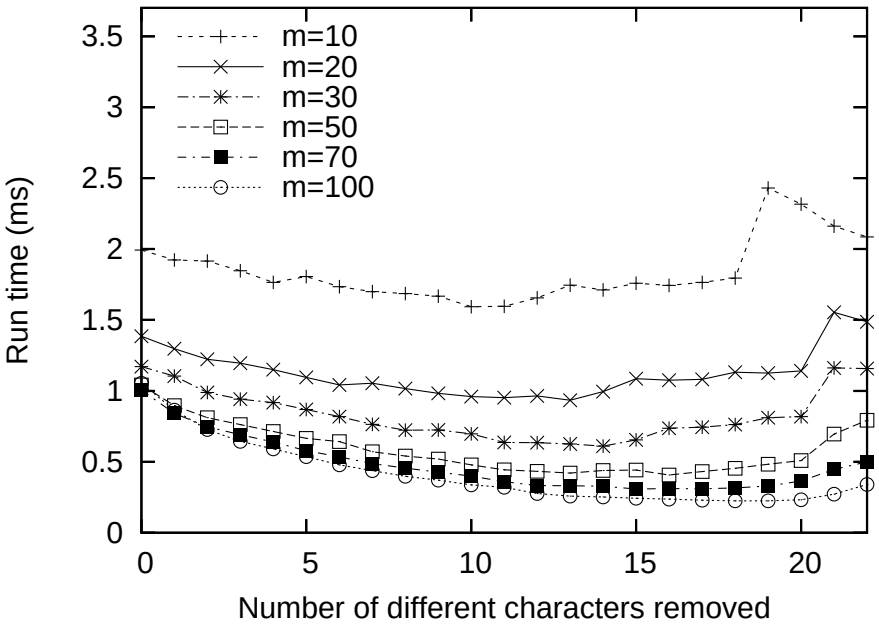


Figure 7.6: Runtime for the tuned version of the sampled semi-index using $f_2(n, m, \bar{\sigma}) = n/\min(m, \bar{\sigma})$

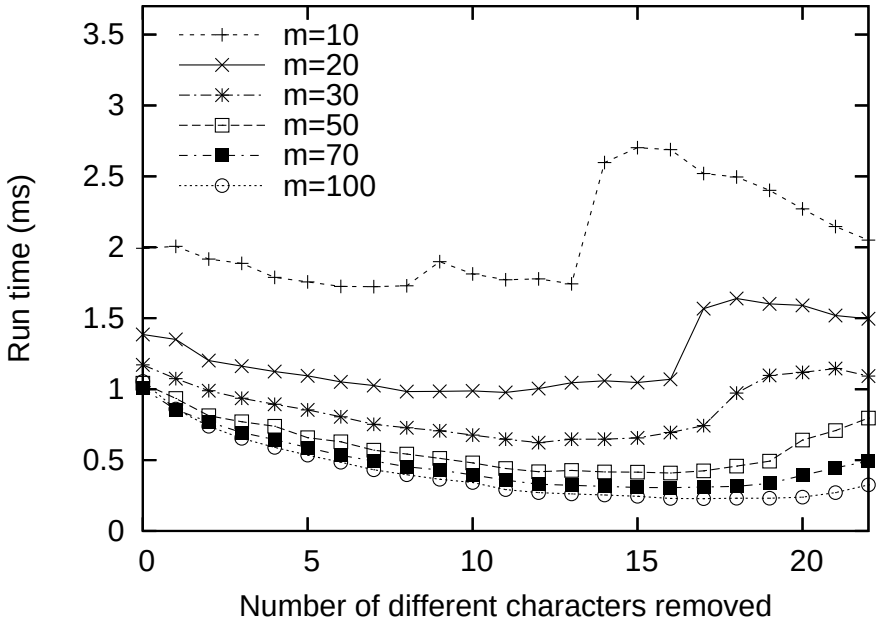


Figure 7.7: Runtime for the tuned version of the sampled semi-index using $f_3(n, m, \bar{\sigma}) = n \cdot (1/m + 1/\bar{\sigma})$

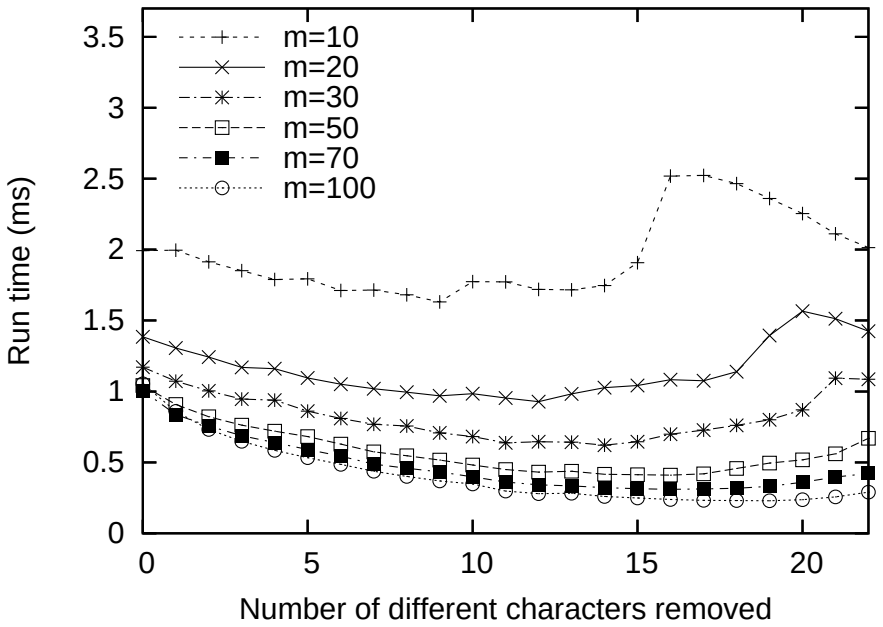


Figure 7.8: Runtime for the tuned version of the sampled semi-index using $f_4(n, \bar{s}) = n/\bar{s}$

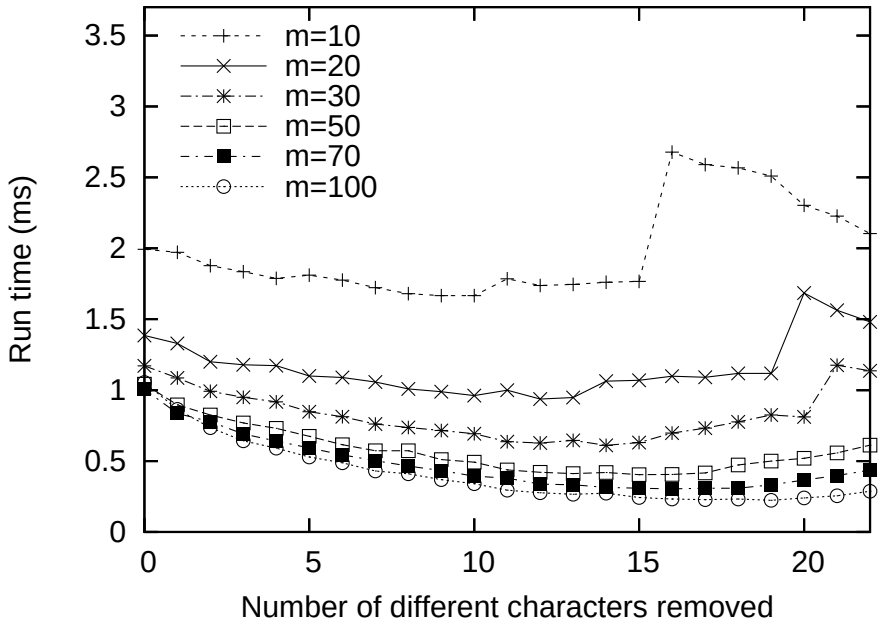


Figure 7.9: Runtime for the tuned version of the sampled semi-index using $f_5(n, m, \bar{\sigma}) = n \cdot (1/m + (m+1)/(2m\bar{\sigma}))$

As we can see, the optimal number of removed characters is closer to being the same for all pattern lengths in the best tuned approach than in the basic approach. For example, by choosing to remove the 13 most frequent characters, we would do reasonably well for all pattern lengths using just 0.18 times the original text size to store the sampled text. Figure 7.10 shows the distribution of the runtime for the best tuned sampled semi-index. Comparing Figures 7.4(b) and 7.10, we see that the median running times are almost the same, but the maximum and the 75% quartile are lower for the tuned method. This is also reflected in the average values.

To further test the results of the analysis, we generated all those sets of removed characters that the exact algorithm of Section 7.3 tries. Out of these, we selected all sets of size at most 20 and ran experiments with the semi-index using those sets. Results for the basic method are shown in Tables B.1 and B.2. The first table shows the best 20 sets of removed characters sorted by runtime, and the second table shows the best 20 sets sorted by the number of characters read. From these results, we can see that removing vowels seems to be somewhat more beneficial than suggested by their frequency, especially when the optimal set of removed characters is small. This is probably due to the alternating structure of vowels and consonants in natural language texts. Results of the same experiments for the best tuned method are shown in Tables B.3 and B.4. For the best tuned method, more characters should be removed, and thus the benefits of removing vowels rather than consonants are not so pronounced.

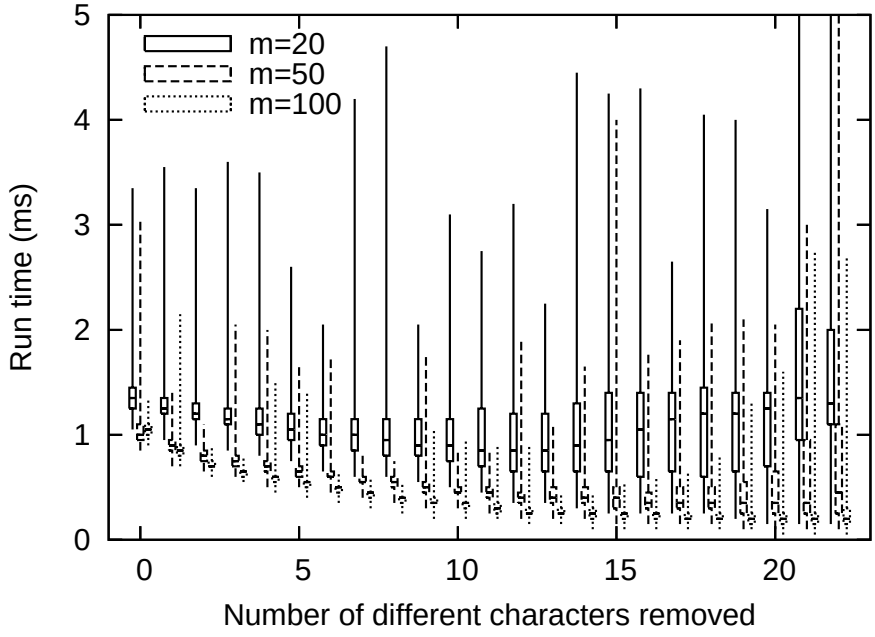


Figure 7.10: The distribution of running time for various pattern lengths for the best tuned sampled semi-index. The figure shows the median, minimum, maximum, and 25% and 75% quartiles.

Chapter 8

Conclusions

We have developed algorithms for several string matching problems using the q-gram backward string matching paradigm. The Boyer-Moore-Horspool algorithm was extended for approximate and parameterized string matching by exploiting q-grams, and two q-gram backward string matching algorithms were developed for multiple string matching. Of these, we showed that the algorithms for exact, approximate, and multiple string matching are optimal on average. The average complexity of parameterized string matching is not known, but we showed that the q-gram backward string matching paradigm results in sublinear average case complexity for a class of moderately repetitive patterns in this case also. Thus, the q-gram backward string matching paradigm proved to be an effective tool to develop string matching algorithms.

Not all average optimal string matching algorithms are q-gram backward string matching algorithms. Fredriksson and Grabowski [39, 40] have recently introduced a family of average optimal algorithms that are not based on the backward matching principle. Their algorithms read every q:th character of the text and verify a position if the read characters indicate that there could be a match at that position.

We carried out extensive experiments to compare the new algorithms with older ones and found the new algorithms to be very competitive in most scenarios. The various experiments on DNA data show that the developed algorithms for approximate string matching, weighted string matching, and multiple string matching are faster than old ones for many search problems on DNA sequences. The new algorithms for multiple string matching also performed very well on random data with alphabet size 256, which is a scenario similar to anti-virus scanning. However, further experiments on real data would be needed to confirm the good performance in real applications.

In string matching problems, the probability of finding an occurrence of the pattern typically decreases exponentially when the pattern length increases. This is crucial for the success of backward string matching because the average number of characters we need to read in a window to deduce that there cannot be a match increases only logarithmically in the length of the pattern. Thus the algorithms can skip larger and larger parts of the text as the pattern length increases.

In the weighted string matching problem, the probability of finding an occurrence of the pattern is fixed by the significance level. This changes the statistics of the string matching problem radically, as increasing the length of a pattern no longer translates into an exponential decrease in the probability of finding an occurrence. From an algorithmic point of view, the average number of characters we need to read to be able to deduce that there cannot be a match at a given position increases linearly in the length of the pattern. Thus, linear average case complexity might be the best we can achieve in many cases for weighted string matching. The experimental results of Chapter 6 support these ideas. The linear time bit parallel algorithms are better than the backward matching ones, and the best backward matching algorithm, eBG, is only competitive for high significance levels and fairly short patterns, whereas in traditional string matching problems, backward matching algorithms typically excel with long patterns.

Another interesting finding of this work is that multiple string matching is a competitive alternative to indexing methods. This is especially true for applications where large sets of patterns arrive at the same time rather than one pattern at a time over a longer period of time. Not only is the time to search the patterns shorter than in indexing methods, but the memory usage of the new multiple string matching algorithms is also moderate, making them very practical.

Exploiting the nonuniform character distribution of real texts is not a new idea in online string matching. For example, Boyer-Moore-Horspool type algorithms which sort the characters of the pattern in increasing order of probability of occurrence in the text and check the characters of a text window in this order have been developed [98]. The idea of the sampled semi-index is similar in spirit although the actual approach is quite different. An interesting area of further work would be to integrate the statistical dependencies of nearby characters into these models.

Bibliography

[1] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

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

[3] A.V. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[4] C. Allauzen and M. Raffinot. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999. (in French).

[5] A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat. Function matching: Algorithms, applications and a lower bound. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, volume 2719 of *LNCS*, pages 929–942. Springer-Verlag, 2003.

[6] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *Information Processing Letters*, 49(3):111–115, 1994.

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

[8] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. On economic construction of the transitive closure of a directed graph. *Doklady Academi Nauk SSSR*, 194:487–488, 1970. (in Russian). English translation in Soviet Mathematics Doklady 11:1209–1210, 1975.

[9] R. Baeza-Yates. Improved string searching. *Software – Practice and Experience*, 19(3):257–271, 1989.

[10] R. Baeza-Yates. String searching algorithms revisited. In *Proceedings of the Workshop on Algorithms and Data Structures (WADS'89)*, volume 382 of *LNCS*, pages 75–96. Springer-Verlag, 1989.

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

[12] R. Baeza-Yates and G.H. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994.

[13] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

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

[15] R. Baeza-Yates and M. Régnier. Fast two-dimensional pattern matching. *Information Processing Letters*, 45(1):51–57, 1993.

[16] B.S. Baker. A theory of parameterized pattern matching: Algorithms and applications (extended abstract). In *Proceedings of the 25th Annual ACM Symposium on the Theory of Computation (STOC'93)*, pages 71–80. ACM Press, 1993.

[17] B.S. Baker. Parameterized pattern matching by Boyer-Moore-type algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'95)*, pages 541–550. SIAM, 1995.

[18] B.S. Baker. Parameterized diff. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, pages 854–855. SIAM, 1999.

[19] T.P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978.

[20] D.R. Bentley. Whole-genome re-sequencing. *Current Opinion in Genetics & Development*, 16(6):545–552, 2006.

[21] T. Berry and S. Ravindran. A fast string matching algorithm and experimental results. In *Proceedings of the Prague Stringology Club Workshop'99*, pages 16–26. Czech Technical University, 1999.

[22] R.S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

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

[24] W.I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proceedings of the 3rd Annual Symposium of Combinatorial Pattern Matching (CPM'92)*, volume 644 of *LNCS*, pages 175–184. Springer-Verlag, 1992.

[25] W.I. Chang and T.G. Marr. Approximate string matching and local similarity. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM'94)*, volume 807 of *LNCS*, pages 259–273. Springer-Verlag, 1994.

[26] F. Claude, G. Navarro, H. Peltola, L. Salmela, and J. Tarhio. Speeding up string matching with text sampling. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE'08)*, volume 5280 of *LNCS*, pages 87–98. Springer-Verlag, 2008.

[27] J.-M. Claverie and S. Audic. The statistical significance of nucleotide position-weight matrix matches. *Computer Applications in the Biosciences*, 12(5):431–439, 1996.

[28] R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. In *Proceedings of the 32nd Annual ACM Symposium on the Theory of Computation (STOC'00)*, pages 407–415. ACM Press, 2000.

[29] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proceedings of the 6th Colloquium on Automata, Languages and Programming (ICALP'79)*, volume 71 of *LNCS*, pages 118–132. Springer-Verlag, 1979.

[30] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12(4–5):247–267, 1994.

[31] M. Crochemore, C.S. Iliopoulos, G. Navarro, Y.J. Pinzon, and A. Salinger. Bit-parallel (δ, γ)-matching and suffix automata. *Journal of Discrete Algorithms*, 3(2–4):198–214, 2005.

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

[33] M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific Press, 2002.

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

[35] N. El-Mabrouk and M. Crochemore. Boyer–Moore strategy to efficient approximate string matching. In *Proceedings of 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, volume 1075 of *LNCS*, pages 24–38. Springer-Verlag, 1996.

[36] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice! Manuscript. `http://pizzachili.dcc.uchile.cl/`, 2007.

[37] M. Fisk and G. Varghese. Fast content-based packet handling for intrusion detection. Technical Report CS2001-0670, University of California, San Diego, 2001.

[38] K. Fredriksson. Shift-or string matching with super-alphabets. *Information Processing Letters*, 87(4):201–204, 2003.

[39] K. Fredriksson and S. Grabowski. Average-optimal string matching. *Journal of Discrete Algorithms*. (In press).

[40] K. Fredriksson and S. Grabowski. Practical and optimal string matching. In *Proceedings of 12th International Conference on String Processing and Information Retrieval (SPIRE'05)*, volume 3772 of *LNCS*, pages 376–387. Springer-Verlag, 2005.

[41] K. Fredriksson and M. Mozgovoy. Efficient parameterized string matching. *Information Processing Letters*, 100(3):91–96, 2006.

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

[43] Z. Galil and K. Park. Truly alphabet-independent two-dimensional pattern matching. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS'92)*, pages 247–256. IEEE, 1992.

[44] M. Gribskov, A.D. McLachlan, and D. Eisenberg. Profile analysis: Detection of distantly related proteins. *Proceedings of the National Academy of Sciences of the United States of America*, 84(13):4355–4358, 1987.

[45] B. Gum and R. Lipton. Cheaper by the dozen: Batched algorithms. In *Proceedings of the 1st SIAM International Conference on Data Mining (SDM'01)*, 2001.

[46] D. Gusfield. *Algorithms on strings, trees and sequences: Computer science and computational biology*. Cambridge University Press, 1997.

[47] C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA'04)*, volume 3221 of *LNCS*, pages 414–425. Springer-Verlag, 2004.

[48] C. Hazay, M. Lewenstein, and D. Tsur. Two dimensional parameterized matching. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05)*, volume 3537 of *LNCS*, pages 266–279. Springer-Verlag, 2005.

[49] R.N. Horspool. Practical fast searching in strings. *Software – Practise and Experience*, 10(6):501–506, 1980.

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

[51] R.M. Idury and A.A. Schäffer. Multiple matching of parameterized patterns. *Theoretical Computer Science*, 154(2):203–224, 1996.

[52] P. Kalsi, L. Salmela, and J. Tarhio. Tuning approximate Boyer-Moore for gene sequences. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE'07)*, volume 4726 of *LNCS*, pages 173–183. Springer-Verlag, 2007.

[53] J. Kärkkäinen and E. Ukkonen. Two and higher dimensional pattern matching in optimal expected time. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 715–723. SIAM, 1994.

[54] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[55] J.E. Kasprzak and M.A. Nixon. Cheating in cyberspace: Maintaining quality in online education. *Association for the Advancement of Computing In Education*, 12(1):85–99, 2004.

[56] S. Kim and Y. Kim. A fast multiple string-pattern matching algorithm. In *Proceedings of 17th AoM/IAoM Conference on Computer Science*, 1999.

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

[58] S.R. Kosaraju. Faster algorithms for the construction of parameterized suffix trees. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*, pages 631–637. IEEE, 1995.

[59] D.L. Kreher and D.R. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC Press, 1999.

[60] J. Kytöjoki, L. Salmela, and J. Tarhio. Tuning string matching for huge pattern sets. In *Proceedings of 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03)*, volume 2676 of *LNCS*, pages 211–224. Springer-Verlag, 2003.

[61] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[62] V.I. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1(1):8–17, 1965.

[63] A. Liefooghe, H. Touzet, and J.-S. Varré. Large scale matching for position weight matrices. In *Proceedings of 17th Annual Symposium on Combinatorial Pattern Matching (CPM'06)*, volume 4009 of *LNCS*, pages 401–412. Springer-Verlag, 2006.

[64] P. Liu, Y.-B. Liu, and J.-L. Tan. A partition-based efficient algorithm for large scale multiple-strings matching. In *Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE'05)*, volume 3772 of *LNCS*, pages 399–404. Springer-Verlag, 2005.

[65] Z. Liu, X. Chen, J. Borneman, and T. Jiang. A fast algorithm for approximate string matching on gene sequences. In *Proceedings of 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05)*, volume 3537 of *LNCS*, pages 79–90. Springer-Verlag, 2005.

[66] R. Lowrance and R.A. Wagner. An extension of the string-to-string correction problem. *Journal of the ACM*, 22(2):177–183, 1975.

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

[68] E.P. Markatos, S. Antonatos, M. Polychronakis, and K.G. Anagnostakis. Exclusion-based signature matching for intrusion detection. In *Proceedings of the IASTED International Conference on Communications and Computer Networks (CCN'02)*, pages 146–152. ACTA Press, 2002.

[69] W.J. Masek and M.S. Paterson. A faster algorithm for computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[70] V. Matys, E. Fricke, R. Geffers, E. Gößling, M. Haubrock, R. Hehl, K. Hornischer, D. Karas, A.E. Kel, O.V. Kel-Margoulis, D.-U. Kloos, S. Land, B. Lewicki-Potapov, H. Michael, R. Münch, I. Reuter, S. Rotert, H. Saxel, M. Scheer, S. Thiele, and E. Wingender. TRANSFAC®: transcriptional regulation, from patterns to profiles. *Nucleic Acids Research*, 31(1):374–378, 2003.

[71] E.S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

[72] R. Muth and U. Manber. Approximate multiple string search. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, volume 1075 of *LNCS*, pages 75–86. Springer-Verlag, 1996.

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

[74] C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, 1997.

[75] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[76] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001.

[77] G. Navarro and K. Fredriksson. Average complexity of exact and approximate multiple string matching. *Theoretical Computer Science*, 321(2–3):283–290, 2004. Errata in `http://www.dcc.uchile.cl/~gnavarro/erratas/tcs04.html`.

[78] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):1–61, 2007.

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

[80] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Text and Biological Sequences*. Cambridge University Press, 2002.

[81] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00)*, volume 1848 of *LNCS*, pages 350–363. Springer-Verlag, 2000.

[82] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[83] C. Pizzi, P. Rastas, and E. Ukkonen. Fast search algorithms for position specific scoring matrices. In *Proceedings of the 1st International Conference on Bioinformatics Research and Development (BIRD'07)*, volume 4414 of *LNBI*, pages 239–250. Springer-Verlag, 2007.

[84] C. Pizzi and E. Ukkonen. Fast profile matching algorithms – a survey. *Theoretical Computer Science*, 395(2–3):137–157, 2008.

[85] T. Raita. Tuning the Boyer-Moore-Horspool string searching algorithm. *Software – Practise and Experience*, 22(10):879–884, 1992.

[86] L. Salmela and J. Tarhio. Sublinear algorithms for parameterized matching. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM'06)*, volume 4009 of *LNCS*, pages 354–364. Springer-Verlag, 2006.

[87] L. Salmela and J. Tarhio. Algorithms for weighted matching. In *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE'07)*, volume 4726 of *LNCS*, pages 276–286. Springer-Verlag, 2007.

[88] L. Salmela and J. Tarhio. Fast parameterized matching with q-grams. *Journal of Discrete Algorithms*, 6(3):408–419, 2008.

[89] L. Salmela, J. Tarhio, and P. Kalsi. Approximate Boyer-Moore string matching for small alphabets. *Algorithmica*. (In press).

[90] L. Salmela, J. Tarhio, and J. Kytöjoki. Multipattern string matching with q-grams. *ACM Journal of Experimental Algorithmics*, 11(1.1):1–19, 2006.

[91] D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences of the United States of America*, 69(1):4–6, 1972.

[92] P.M. Scanlon and D.R. Neumann. Internet plagiarism among college students. *Journal of College Student Development*, 43(3):374–385, 2002.

[93] T.D. Scheiner, G.D. Stormo, L. Gold, and A. Ehrenfeucht. Information content of binding sites on nucleotide sequences. *Journal of Molecular Biology*, 188(3):415–431, 1986.

[94] P.H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

[95] P.H. Sellers. The theory and computation of evolutionary distances: Pattern recognition. *Journal of Algorithms*, 1(4):359–373, 1980.

[96] P.D. Smith. Experiments with a very fast substring search algorithm. *Software – Practise and Experience*, 21(10):1065–1074, 1991.

[97] R. Staden. Methods for calculating the probabilities of finding patterns in sequences. *Computer Applications in the Biosciences*, 5(2):89–96, 1989.

[98] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, 1990.

[99] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proceedings of the 3rd Annual European Symposium on Algorithms (ESA'95)*, volume 979 of *LNCS*, pages 327–340. Springer-Verlag, 1995.

[100] J. Tarhio. A sublinear algorithm for two-dimensional string matching. *Pattern Recognition Letters*, 17(8):833–838, 1996.

[101] J. Tarhio and E. Ukkonen. Approximate Boyer–Moore string matching. *SIAM Journal on Computing*, 22(2):243–260, 1993.

[102] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, volume 4, pages 2628–2639, 2004.

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

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

[105] T.K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics and System Analysis*, 4(1):52–57, 1968.

[106] R.A. Wagner and M.J. Fisher. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[107] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.

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

[109] S. Wu, U. Manber, and G. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

[110] T.D. Wu, C.G. Nevill-Manning, and D.L. Brutlag. Fast probabilistic analysis of sequence function using scoring matrices. *Bioinformatics*, 16(3):233–244, 2000.

[111] A.C.-C. Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979.

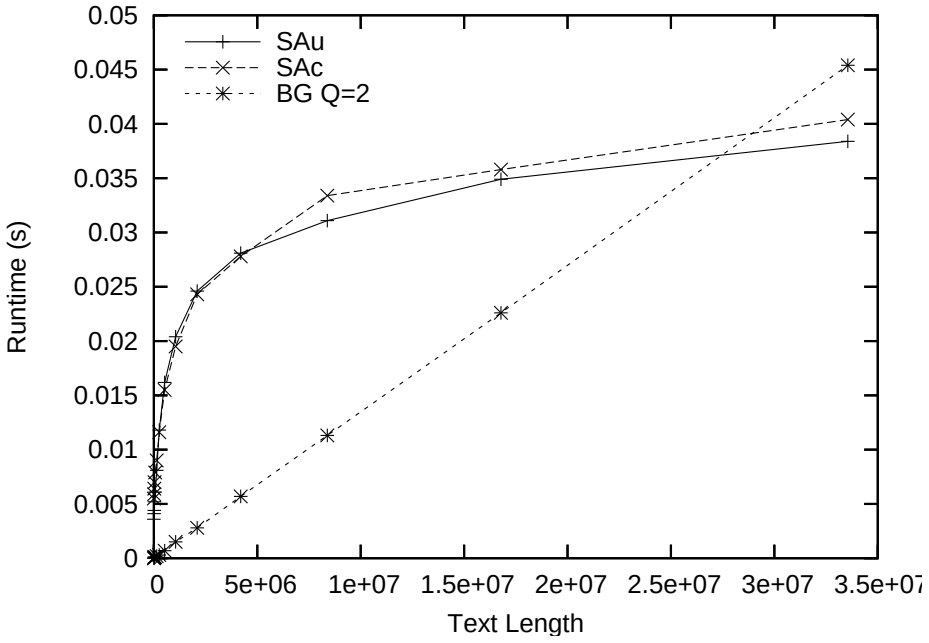
[112] Z. Zhou, Y. Xue, J. Liu, W. Zhang, and J. Li. MDH: A high speed multi-phase dynamic hash string matching algorithm for large-scale pattern set. In *Proceedings of the 9th International Conference on Information and Communications Security (ICICS'07)*, volume 4861 of *LNCS*, pages 201–215. Springer-Verlag, 2007.

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

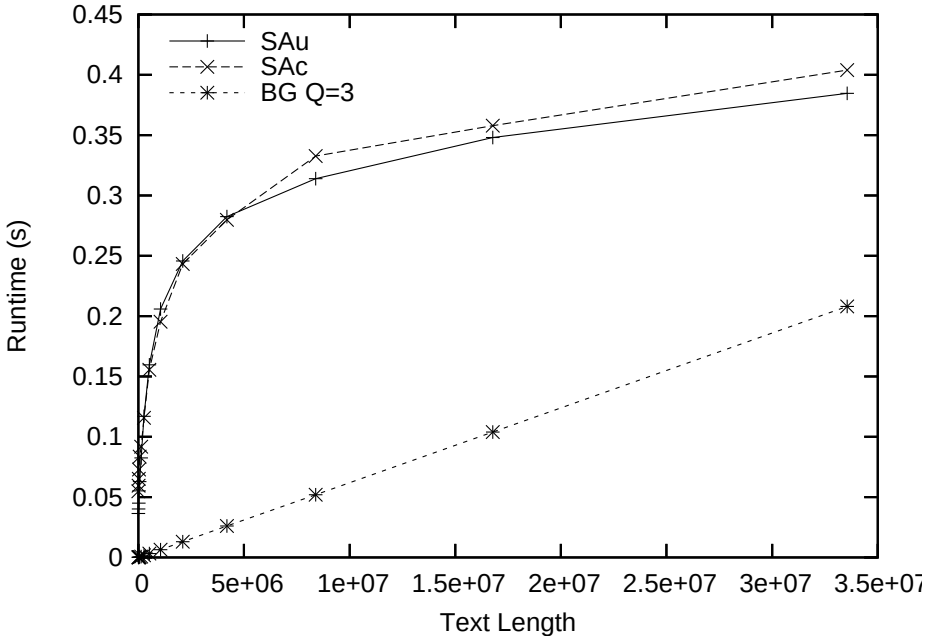
[114] R.F. Zhu and T. Takaoka. A technique for two-dimensional pattern matching. *Communications of the ACM*, 32(9):1110–1120, 1989.

Appendix A

Comparison of the Suffix Array and the BG Algorithm

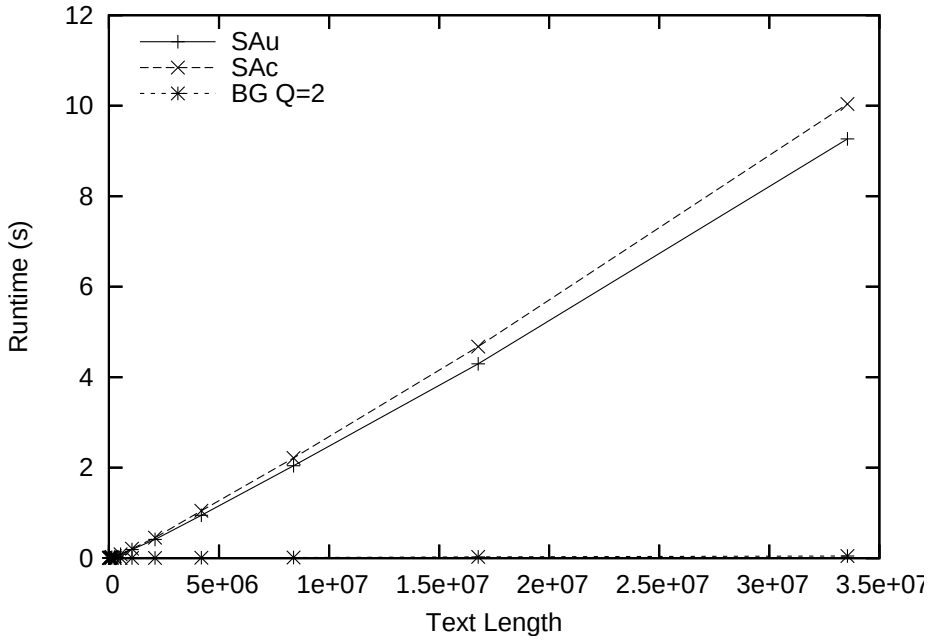


(a) 10,000 patterns

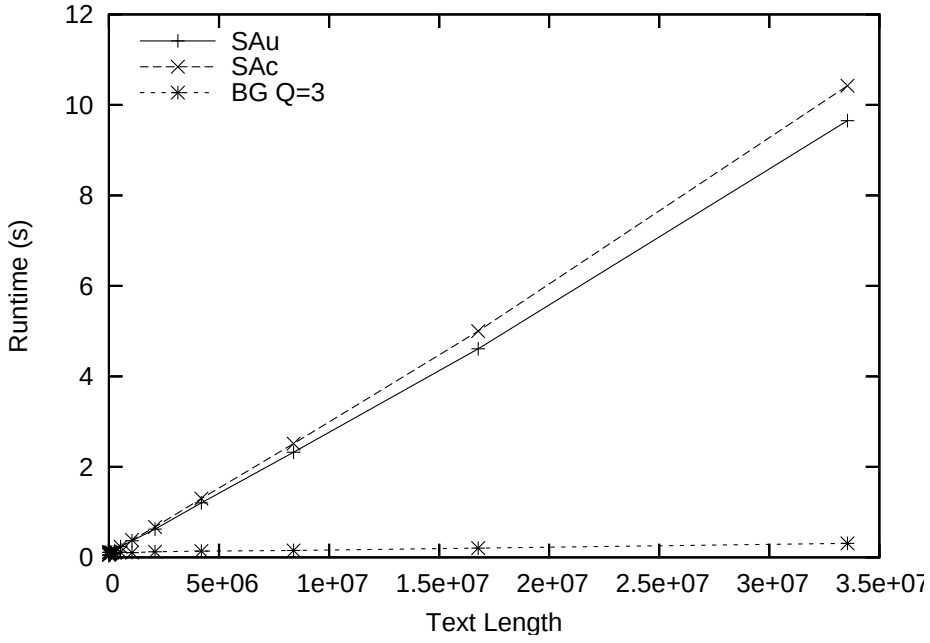


(b) 100,000 patterns

Figure A.1: Search times of BG and suffix array with random data of alphabet size 255

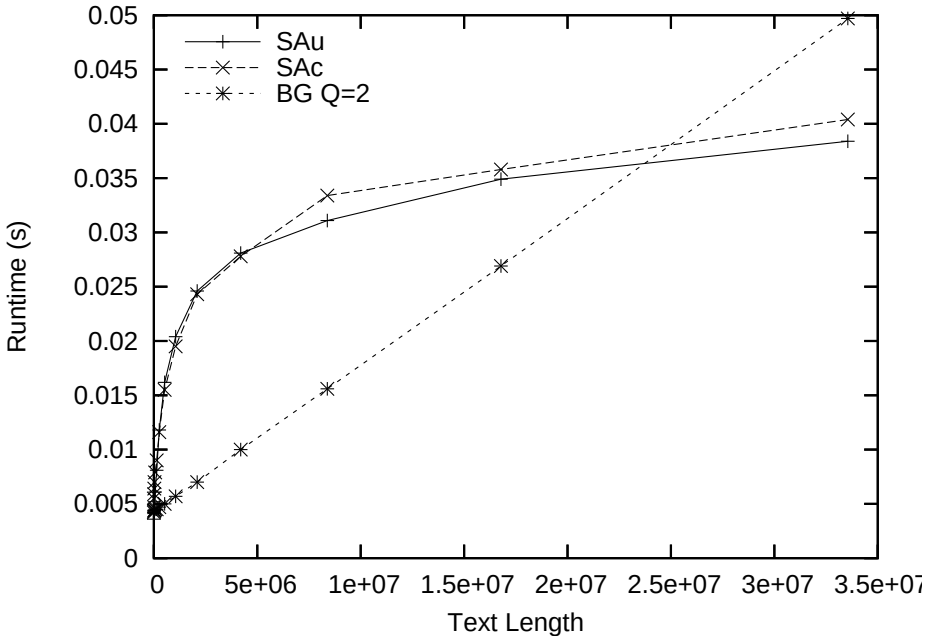


(a) 10,000 patterns

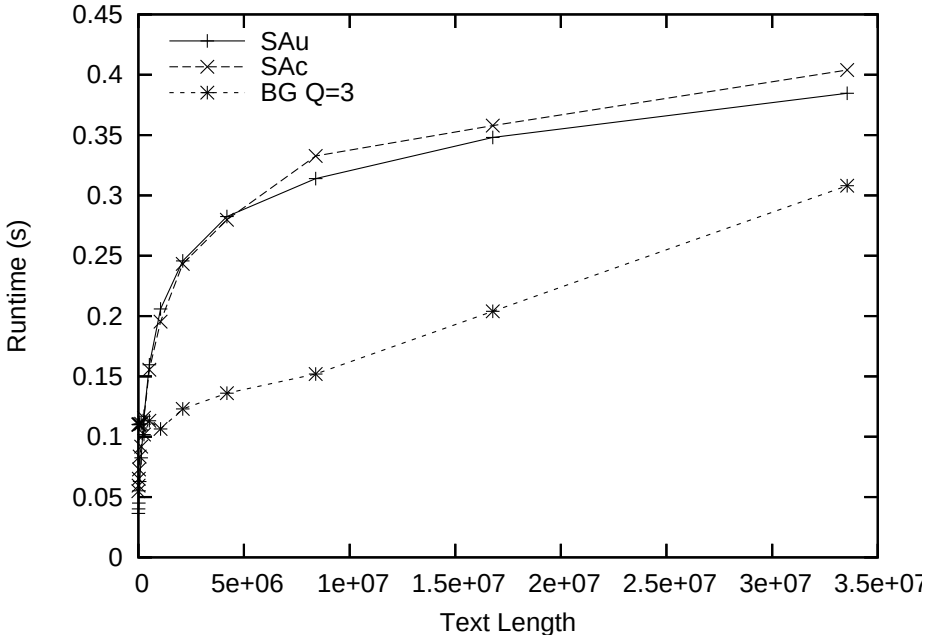


(b) 100,000 patterns

Figure A.2: Combined preprocessing and search times of BG and suffix array with random data of alphabet size 255

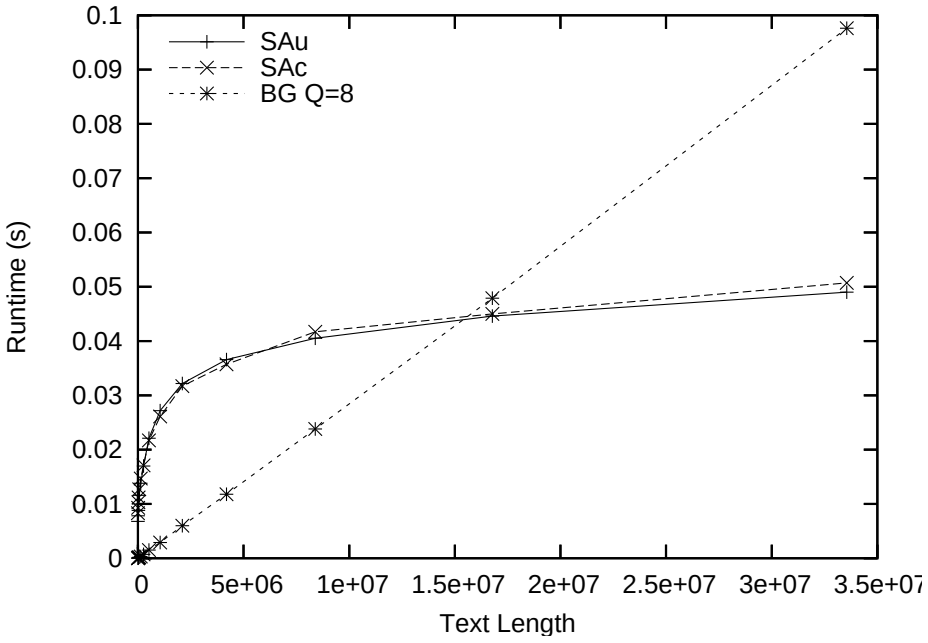


(a) 10,000 patterns

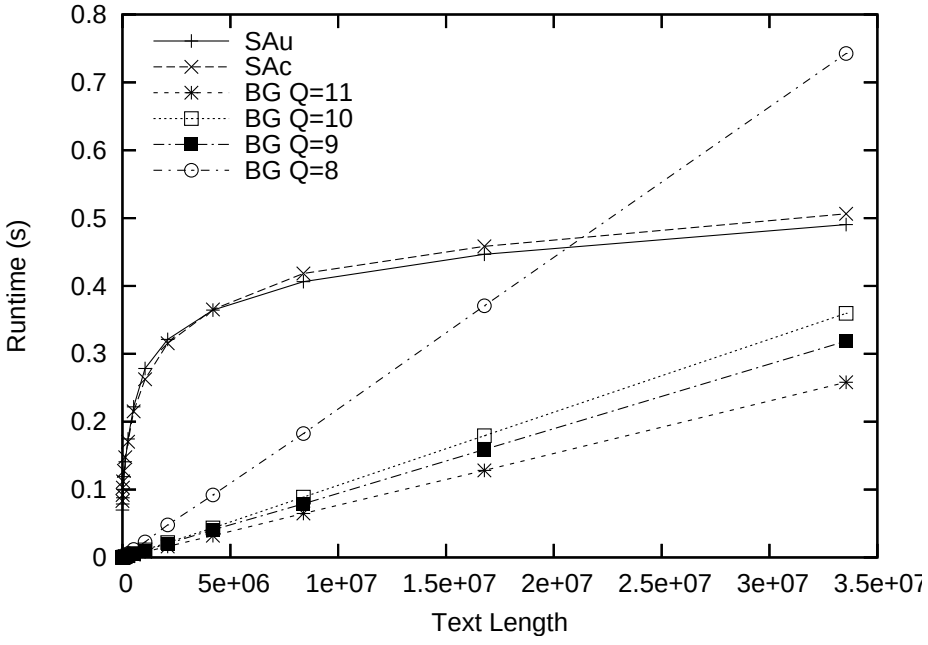


(b) 100,000 patterns

Figure A.3: Combined preprocessing and search times of BG and search times for the suffix array with random data of alphabet size 255

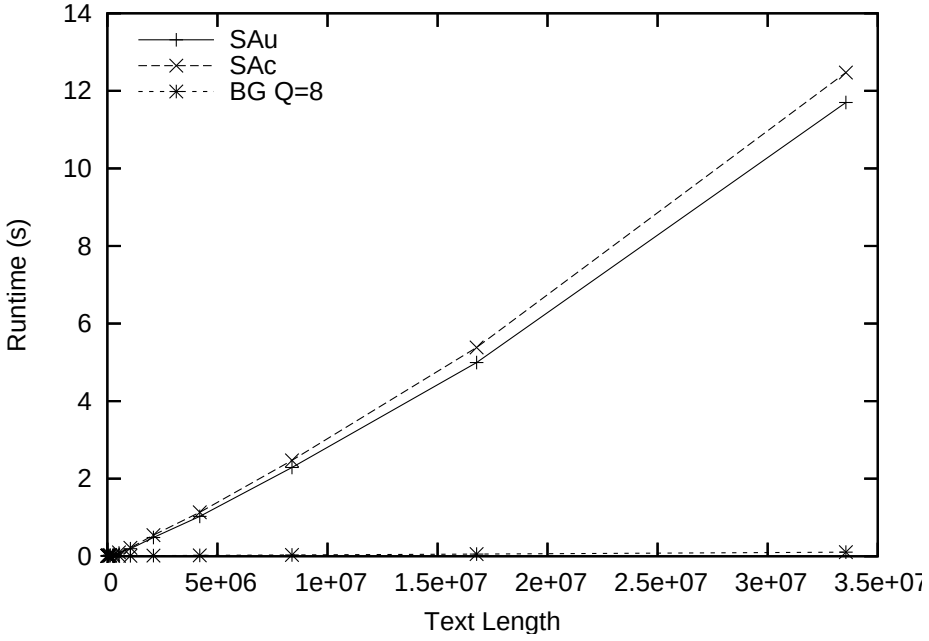


(a) 10,000 patterns

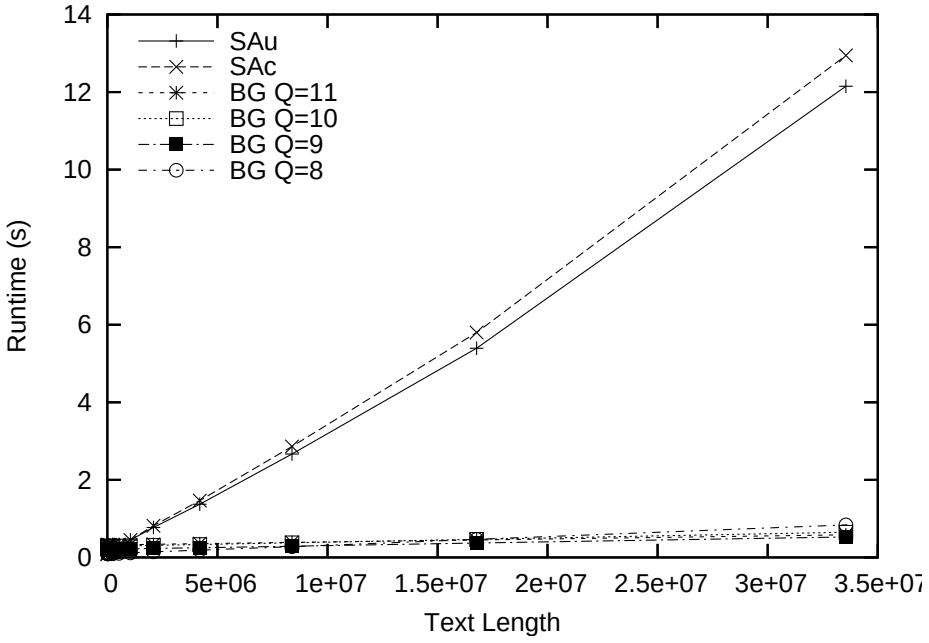


(b) 100,000 patterns

Figure A.4: Search times of BG and suffix array with DNA data

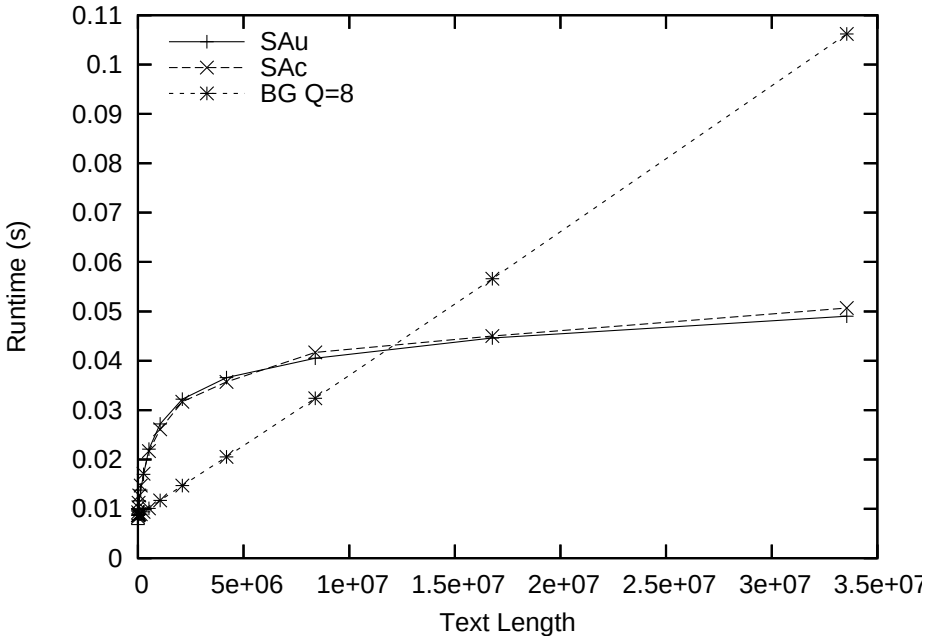


(a) 10,000 patterns

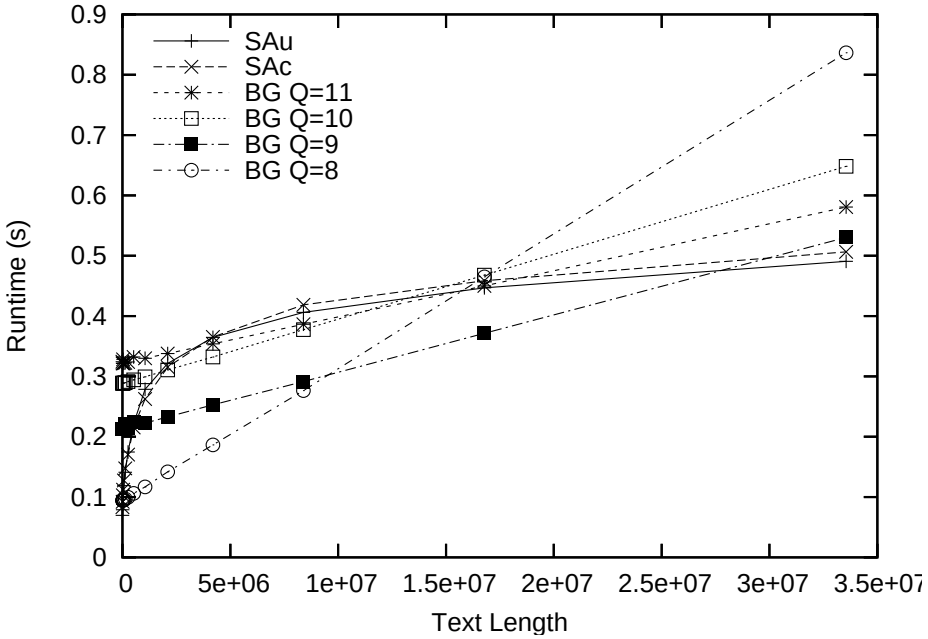


(b) 100,000 patterns

Figure A.5: Combined preprocessing and search times of BG and suffix array with DNA data



(a) 10,000 patterns



(b) 100,000 patterns

Figure A.6: Combined preprocessing and search times of BG and search times for the suffix array with DNA data

Appendix B

Experiments with the Sampled Semi-Index

Table B.1: The best sets of removed characters for the basic method sorted by runtime. The first row shows the runtimes of the Boyer-Moore-Horspool algorithm.

	$m = 10$		$m = 20$		$m = 30$		$m = 50$	
	Removed Set	Runtime (s)	Removed Set	Runtime (s)	Removed Set	Runtime (s)	Removed Set	Runtime (s)
		1.0035		0.6891		0.5897		0.5127
1.	␣eho	0.9337	␣ethaons	0.4978	␣ethaonsirdly	0.3202	␣ethaonsirdluw	0.2029
2.	␣eaoi	0.9363	␣ethaon	0.5004	␣ethaonsird	0.3235	␣ethaonsirdlumc	0.2036
3.	␣eha	0.9364	␣ethaonsr	0.5011	␣ethaonsirdum	0.3269	␣ethaonsirdlfu,w	0.2044
4.	␣ehai	0.9366	␣ethaonr	0.5045	␣ethaonsirdluy	0.3286	␣ethaonsirdlfmw	0.2049
5.	␣ehao	0.9373	␣ethaonid	0.5049	␣ethaonsird,	0.3303	␣ethaonsirdl	0.2052
6.	␣ehaon	0.9387	␣ethaoni	0.5050	␣ethaonsrdl	0.3308	␣ethaonsirdlm,	0.2054
7.	␣eao	0.9414	␣ethaonir	0.5072	␣ethaonsirdlu	0.3315	␣ethaonsirdl,	0.2055
8.	␣eta	0.9417	␣ethaonirl	0.5077	␣ethaonsirdm	0.3317	␣ethaonsirdlum	0.2057
9.	␣eo	0.9426	␣ethaonsd	0.5086	␣ethaonsirlu	0.3330	␣ethaonsirdlfu	0.2059
10.	␣ean	0.9427	␣ethaonidl	0.5086	␣ethaonsirdl,	0.3333	␣ethaonsirdl,c	0.2060
11.	␣eaos	0.9429	␣ethaonsl	0.5092	␣ethaonsidlu	0.3336	␣ethaonsirdlmy	0.2062
12.	␣etai	0.9444	␣ethaonl	0.5112	␣ethaonsirdl,y	0.3337	␣ethaonsirdlfum	0.2063
13.	␣etoi	0.9467	␣ethaonsi	0.5131	␣ethaonsirdfu	0.3344	␣ethaonsirdlu	0.2064
14.	␣ea	0.9468	␣ethaonrl	0.5136	␣ethaonsirdu	0.3347	␣ethaonsirdl,w	0.2067
15.	␣et	0.9475	␣ethaonil	0.5142	␣ethaonirdlu	0.3355	␣ethaonsirdlm	0.2067
16.	␣etn	0.9484	␣ethaoird	0.5163	␣ethaonirdl	0.3361	␣ethaonsirdlm,g	0.2069
17.	␣ets	0.9489	␣ethaondl	0.5176	␣ethaonsiru	0.3370	␣ethaonsirdlfu,	0.2069
18.	␣ehni	0.9492	␣ethaonsir	0.5177	␣ethaonsirdlc	0.3377	␣ethaonsirdluc	0.2069
19.	␣etaod	0.9498	␣ethaoidl	0.5188	␣ethaonsrdlu	0.3378	␣ethaonsirdlf	0.2069
20.	␣eai	0.9501	␣ethaoir	0.5191	␣ethaonsirdlf	0.3384	␣ethaonsirdlumw	0.2070

Table B.2: The best sets of removed characters for the basic method sorted by the number of read characters. The first row shows the number of read characters for the Boyer-Moore-Horspool algorithm.

	$m = 10$			$m = 20$			$m = 30$			$m = 50$		
	Removed Set	Reads	%	Removed Set	Reads	%	Removed Set	Reads	%	Removed Set	Reads	%
		589012.1	29.5		377826.2	18.9		303274.0	15.2		239260.6	12.0
1.	␣ehaoi	502199.2	25.1	␣ethaoni	268311.5	13.4	␣ethaonsirdly	175791.4	8.8	␣ethaonsirdlum	111070.3	5.6
2.	␣eaoi	503864.7	25.2	␣ethaonil	270073.7	13.5	␣ethaonsird	176768.0	8.8	␣ethaonsirdlumw	111326.9	5.6
3.	␣ehao	504863.3	25.2	␣ethaonirl	270395.8	13.5	␣ethaonsirdl	176831.8	8.8	␣ethaonsirdlu	111485.5	5.6
4.	␣etaoi	506216.1	25.3	␣ethaonir	270600.9	13.5	␣ethaonsirdu	177090.2	8.9	␣ethaonsirdlu,	111631.2	5.6
5.	␣eao	506643.8	25.3	␣ethaon	270600.9	13.5	␣ethaonsirdluy	178012.9	8.9	␣ethaonsirdlmw	111678.0	5.6
6.	␣etao	507703.5	25.4	␣ethaons	271730.7	13.6	␣ethaonsirdlu	178846.6	8.9	␣ethaonsirdlm	111699.2	5.6
7.	␣ehaon	509085.0	25.5	␣ethaonl	271900.7	13.6	␣ethaonsirl	179679.1	9.0	␣ethaonsirdlm,	111843.9	5.6
8.	␣etai	509224.1	25.5	␣ethaonr	272520.1	13.6	␣ethaonsirdum	180054.8	9.0	␣ethaonsirdluw	111912.9	5.6
9.	␣eoi	509398.3	25.5	␣ethaonid	272651.2	13.6	␣ethaonsirlu	180327.8	9.0	␣ethaonsirdlm,w	112071.3	5.6
10.	␣ehai	509512.3	25.5	␣ethaonsl	273243.7	13.7	␣ethaonsrdl	180642.4	9.0	␣ethaonsirdlumc	112133.1	5.6
11.	␣eta	509874.5	25.5	␣ethaonrl	273986.5	13.7	␣ethaonsrd	180705.6	9.0	␣ethaonsirdl,	112259.3	5.6
12.	␣eho	509900.6	25.5	␣ethaonsi	274152.1	13.7	␣ethaonsir	180851.8	9.0	␣ethaonsirdluc	112360.1	5.6
13.	␣ehoi	509928.4	25.5	␣ethaond	274162.6	13.7	␣ethaonsiru	180949.5	9.0	␣ethaonsirdl	112362.6	5.6
14.	␣eai	510052.3	25.5	␣ethaonsr	274808.6	13.7	␣ethaonsrdu	181352.4	9.1	␣ethaonsirdlum,	112422.5	5.6
15.	␣eha	510082.4	25.5	␣ethaoid	275392.2	13.8	␣ethaonsirdm	181365.8	9.1	␣ethaonsirdlmc	112487.3	5.6
16.	␣ehan	510135.4	25.5	␣ethaoi	275985.2	13.8	␣ethaonirdlu	181382.9	9.1	␣ethaonsirdlw	112571.0	5.6
17.	␣eo	510301.8	25.5	␣ethaondl	276403.7	13.8	␣ethaonsirdlug	181725.7	9.1	␣ethaonsirdlfu	112603.3	5.6
18.	␣ehn	510571.7	25.5	␣ethaonsd	276608.3	13.8	␣ethaonsirdlm	181772.5	9.1	␣ethaonsirdlu,c	112642.6	5.6
19.	␣ehon	510919.5	25.5	␣ethaonsrl	276647.5	13.8	␣ethaonsirdl,	181814.1	9.1	␣ethaonsirdl,w	112671.6	5.6
20.	␣ea	510920.8	25.5	␣ethaonidl	277123.7	13.9	␣ethaonirdl	181998.2	9.1	␣ethaonsirdlmwg	112682.2	5.6

Table B.3: The best sets of removed characters for the best tuned method sorted by runtime. The first row shows the runtime of the Boyer-Moore-Horspool algorithm.

	$m = 10$		$m = 20$		$m = 30$		$m = 50$	
	Removed Set	Runtime (s)	Removed Set	Runtime (s)	Removed Set	Runtime (s)	Removed Set	Runtime (s)
		1.0035		0.6891		0.5897		0.5127
1.	␣ethaonsirf	0.8018	␣ethaonsirdl	0.4593	␣ethaonsirdluy	0.3006	␣ethaonsirdlfu,	0.2035
2.	␣ethaonsif	0.8024	␣ethaonsirdlf	0.4599	␣ethaonsirdly	0.3024	␣ethaonsirdlu,c	0.2042
3.	␣ethaonsid	0.8043	␣ethaonsirdlc	0.4618	␣ethaonsirdlug	0.3063	␣ethaonsirdlumy	0.2044
4.	␣ethaonsirl	0.8046	␣ethaonsirdlw	0.4642	␣ethaonsirdluyg	0.3065	␣ethaonsirdlum,	0.2045
5.	␣ethaonsirm	0.8047	␣ethaonsirdlyc	0.4648	␣ethaonsirdl	0.3068	␣ethaonsirdlum	0.2046
6.	␣ethaonsidu	0.8056	␣ethaonsirl,	0.4659	␣ethaonsirdlu	0.3069	␣ethaonsirdlfm	0.2046
7.	␣ethaonsir	0.8063	␣ethaonsirdl,w	0.4663	␣ethaonsirdluwy	0.3075	␣ethaonsirdlu,	0.2046
8.	␣ethaonsiru	0.8066	␣ethaonsirduw	0.4664	␣ethaonsirdlm	0.3090	␣ethaonsirdlm,w	0.2046
9.	␣ethaonsirdm	0.8078	␣ethaonsirdm,	0.4666	␣ethaonsirdluw	0.3098	␣ethaonsirdlmc	0.2060
10.	␣ethaonsirfm	0.8086	␣ethaonsird,w	0.4668	␣ethaonsirdlw	0.3100	␣ethaonsirdluwc	0.2064
11.	␣ethaonsirlf	0.8093	␣ethaonsirdfu	0.4674	␣ethaonsirdlmy	0.3106	␣ethaonsirdlm	0.2064
12.	␣ethaonsrd	0.8093	␣ethaonsirdlmw	0.4678	␣ethaonsirdlyc	0.3111	␣ethaonsirdlfumw	0.2065
13.	␣ethaonsilf	0.8101	␣ethaonsirdlfg	0.4681	␣ethaonsirdlum	0.3114	␣ethaonsirdlf,w	0.2066
14.	␣ethaonsrdu	0.8107	␣ethaonsirdlfy	0.4688	␣ethaonsirdlf	0.3118	␣ethaonsirdlu,g	0.2067
15.	␣ethaonsidm	0.8112	␣ethaonsirdlmc	0.4693	␣ethaonsirdlu,y	0.3119	␣ethaonsirdlm,y	0.2068
16.	␣ethaonsird	0.8117	␣ethaonsirdl,	0.4698	␣ethaonsirdluyc	0.3120	␣ethaonsirdl,w	0.2068
17.	␣ethaonsirdu	0.8122	␣ethaonsirdfm	0.4700	␣ethaonsirdlumw	0.3121	␣ethaonsirdluwg	0.2069
18.	␣ethaonsidl	0.8135	␣ethaonsirdluy	0.4703	␣ethaonsirdlc	0.3122	␣ethaonsirdlmwg	0.2071
19.	␣ethaonirdu	0.8138	␣ethaonsirdu	0.4711	␣ethaonsirdlm,	0.3127	␣ethaonsirdlf,	0.2071
20.	␣ethaonsirlu	0.8139	␣ethaonsirlm	0.4713	␣ethaonsirdlfu	0.3131	␣ethaonsirdlfum	0.2072

Table B.4: The best sets of removed characters for the best tuned method sorted by the number of read characters. The first row shows the number of read characters for the Boyer-Moore-Horspool algorithm.

	$m = 10$			$m = 20$			$m = 30$			$m = 50$		
	Removed Set	Reads	%	Removed Set	Reads	%	Removed Set	Reads	%	Removed Set	Reads	%
		589012.1	29.5		377826.2	18.9		303274.0	15.2		239260.6	12.0
1.	␣ethaonsif	466642.9	23.3	␣ethaonsirdl	257726.2	12.9	␣ethaonsirdluy	167283.0	8.4	␣ethaonsirdlumw	110696.4	5.5
2.	␣ethaonsir	466700.9	23.3	␣ethaonsirdly	258064.7	12.9	␣ethaonsirdlu	167969.3	8.4	␣ethaonsirdlum	110847.0	5.5
3.	␣ethaonirl	470805.9	23.5	␣ethaonsird,	259493.2	13.0	␣ethaonsirdly	169315.5	8.5	␣ethaonsirdlu,	111173.7	5.6
4.	␣ethaonsirf	471239.1	23.6	␣ethaonsirdl,	259700.8	13.0	␣ethaonsirdlum	169606.1	8.5	␣ethaonsirdluw	111425.0	5.6
5.	␣ethaonsil	471548.1	23.6	␣ethaonsirl,	259899.6	13.0	␣ethaonsirdlumy	169802.2	8.5	␣ethaonsirdlum,	111453.3	5.6
6.	␣ethaonsid	472275.5	23.6	␣ethaonsirdlw	260144.5	13.0	␣ethaonsirdluyg	169926.7	8.5	␣ethaonsirdlu,w	111466.4	5.6
7.	␣ethaonsiru	472520.5	23.6	␣ethaonsirdlm	260353.9	13.0	␣ethaonsirdluwy	169965.5	8.5	␣ethaonsirdlu	111485.5	5.6
8.	␣ethaonsirm	472863.7	23.6	␣ethaonsirdf	260400.1	13.0	␣ethaonsirdlug	170130.9	8.5	␣ethaonsirdlum,w	111546.3	5.6
9.	␣ethaonsrl	473262.1	23.7	␣ethaonsirdlf	260565.9	13.0	␣ethaonsirdl	170298.2	8.5	␣ethaonsirdlmw	111678.0	5.6
10.	␣ethaonird	473641.5	23.7	␣ethaonsirdm	260783.0	13.0	␣ethaonsirdluw	170431.9	8.5	␣ethaonsirdlm	111699.2	5.6
11.	␣ethaosird	474177.8	23.7	␣ethaonsird,w	260876.0	13.0	␣ethaonsirdluc	170566.7	8.5	␣ethaonsirdlm,	111843.9	5.6
12.	␣ethaonsilf	474213.2	23.7	␣ethaonsir	261077.0	13.1	␣ethaonsirdluyc	170625.1	8.5	␣ethaonsirdlumc	111946.1	5.6
13.	␣ethaonirlf	475341.5	23.8	␣ethaonsirdl,y	261171.7	13.1	␣ethaonsirdlu,	170757.5	8.5	␣ethaonsirdlfu	112070.6	5.6
14.	␣ethaonsrlf	475813.2	23.8	␣ethaonsirdlc	261283.0	13.1	␣ethaonsirdlu,y	170855.7	8.5	␣ethaonsirdlm,w	112071.3	5.6
15.	␣ethaonsrd	476412.3	23.8	␣ethaonsirlm	261463.7	13.1	␣ethaonsirdlfu	170883.6	8.5	␣ethaonsirdlfu,	112077.0	5.6
16.	␣ethonsird	476560.2	23.8	␣ethaonsirdm,	261600.5	13.1	␣ethaonsirdlmy	170901.0	8.5	␣ethaonsirdlumb	112184.6	5.6
17.	␣ethaonsidm	477168.1	23.9	␣ethaonsirdmw	261624.5	13.1	␣ethaonsirdlwy	171098.9	8.6	␣ethaonsirdlumg	112189.4	5.6
18.	␣ethaonsilu	477439.5	23.9	␣ethaonsil	261732.0	13.1	␣ethaonsirdlm	171616.0	8.6	␣ethaonsirdlu,c	112197.7	5.6
19.	␣ethaosirdu	478140.1	23.9	␣ethaonsirdu	261966.8	13.1	␣ethaonsirdlfy	171921.3	8.6	␣ethaonsirdlumy	112209.0	5.6
20.	␣ethaon	478271.8	23.9	␣ethaonsirlm,	261981.1	13.1	␣ethaonsirdl,y	172002.7	8.6	␣ethaonsirdl,	112259.3	5.6