# Approximate Text Searching

by

**Gonzalo Navarro**

A Thesis presented to the University of Chile
in fulfillment of the thesis requirement
to obtain the degree of
**PhD. in Computer Science**

| | | |
|---|---|---|
| Advisor | : | **Ricardo Baeza-Yates** |
| Committee | : | Jorge Olivos |
| | : | Patricio Poblete |
| | : | Esko Ukkonen (External Professor, Univ. of Helsinki, Finland) |

This work has been supported in part by Fondecyt (Chile) grants 1-950622 and 1-960881, Fondef (Chile) grant 96-1064 and CYTED VII.13 AMYRI Project.

**Dept. of Computer Science - University of Chile**
**Santiago - Chile**
**December 1998**

# Abstract

This thesis focuses on the problem of text retrieval allowing errors, also called "approximate" string matching. The problem is to find a pattern in a text, where the pattern and the text may have "errors". This problem has received a lot of attention in recent years because of its applications in many areas, such as information retrieval, computational biology and signal processing, to name a few.

The aim of this work is the development and analysis of novel algorithms to deal with the problem under various conditions, as well as a better understanding of the problem itself and its statistical behavior. Although our results are valid in many different areas, we focus our attention on typical text searching for information retrieval applications. This makes some ranges of values for the parameters of the problem more interesting than others.

We have divided this presentation in two parts. The first one deals with on-line approximate string matching, i.e. when there is no time or space to preprocess the text. These algorithms are the core of off-line algorithms as well. On-line searching is the area of the problem where better algorithms existed. We have obtained new bounds for the probability of an approximate match of a pattern in a random text, and used these results to analyze many old and new algorithms. We have developed new algorithms for this problem which are currently among the fastest known ones, being even the fastest algorithms for almost all the interesting cases of typical text searching. Finally, we extended our results to the simultaneous search of multiple patterns, obtaining the best existing algorithms when a moderate number of them is sought (less than 100, approximately).

The second part of this thesis addresses indexed approximate string matching, i.e. when we are able to build an index for the text beforehand, to speed up the search later. The ultimate index for approximate string matching is yet to appear and the current development is rather immature, but we have made progress regarding new algorithms as well as better understanding of the problem. For the restricted case of indices able to retrieve only whole words on natural language text, we have obtained new analytical results on their asymptotic complexity, which allowed us to develop an index that is sublinear in space and query time simultaneously, something that did not exist before. For this kind of index we also presented improved search algorithms. For general indices able to find any occurrence (not only words), we have developed new indexing schemes which are a tradeoff between efficiency and space requirements. Also, inspired in on-line techniques, we have proposed a hybrid between existing indexing schemes and obtained very promising results.

It is worth to mention that in almost all cases we have complemented the development of the new algorithms with their worst-case and average-case complexity analysis, as well as a thorough experimental validation and comparison against the best previous work we were aware of.

As a whole, we believe that this work constitutes a valuable contribution to the development and understanding of the problem of approximate text searching.

# Acknowledgments

Lazyness is a widespread sin, and I have been guilty of it when I took the file of the acknowledgments of my MSc. thesis (1995), thinking on modifying it for my PhD. thesis. My surprise was that most of what I was thinking to say was already there, valid in 1995 and valid now, finishing 1998. I am still indebted to the same people I was indebted to by that time, and the debt keeps growing. So if you, reader, have seen the acknowledgments of my MSc. thesis, this is not a careless copy of it, it is a renowed expression of gratitude.

The completion of this thesis does not only mean finishing the most important and ample work I have ever attempted. It also signs the end of a 5 years long stage of my life, an incredibly enjoyable time of endless curiosity, exciting work, and great satisfactions and rewards. It has been also a stage of working in what I like most, with little pressure or interference from other problems or requirements. Fortunately, nothing makes me think that which follows should be very different. What I have been (and hope to keep) enjoying is a researcher life, no more, no less. This is the only life I believe I am able to live.

Therefore, I cannot be less than indebted to all the people which made this kind of life a possibility for me, and I would like to express my gratitude to their sincere and disinterested help and friendship. I am afraid I will forget some names anyway, and I hope they will forgive such a mistake.

First of all, my wife Betina, that left everything to follow me in this adventure. Only she is responsible of keeping me still living a life outside my office. Without she, I would have finished this thesis in half the time, but my life would be by far less interesting to live.

My wife apart, one of the first persons I should mention is Jorge Olivos, who knew me at ESLAI, and has been pushing me since then to leave my job and come to Chile, to enjoy serious research. He also made the first steps in giving me an opportunity to come, and has been always ready to bring me his friendship and help.

I want also to acknowledge Patricio Poblete for his friendship and unconditional support (financial and academic, among others), for the stimulating lectures and joint work I enjoyed with him, and lastly, for being the Department's head with a style totally opposite to what such an administrative position could make one to expect. His accessibility, flexibility and willingness prevented more than one headache at the critical moments.

Ricardo Baeza-Yates, my thesis advisor, deserves a special acknowledgment. This is not only for trusting me from the first time and giving me access to all what he thought I deserved despite my student status, but also for having done an excellent job as my thesis advisor (being there when I needed advise, not intervening when he thought I could manage it) and in general as my guide from being a disoriented student that came to Chile with little idea of what to do, to the end of my studentship and my birth as a new researcher with some idea of what he wants and how far can he go. Not happy with that, he also took care of financially supporting my research, helped me in so

many ways that I have forgotten, and has been a permanent, patient and disinterested friend.

Along these years I have met new people from outside which became my friends too, and I want also to thank them: Nivio Ziviani, Edleno de Moura, Mathieu Raffinot, Edgar Chávez, Jesús Vegas, Marcio Drumond Araújo, João Paulo Kitajima, Berthier Ribeiro, Erkki Sutinen, Gene Myers, Amihood Amir, Pablo de la Fuente, Esteban Feuerstein, and others that I surely forgot to mention.

I would also like to thank all the rest of the people of the Department who made my life easier and more enjoyable, as well as my old friends that are still with me by e-mail and with who I have enjoyed a lot: Sergio Servetto, Guillermo Alvarez, Pablo Martínez-López, ... and of course my family, who is always proud of me, even when I do nothing remarkable. A special note to Pablo Palma, a very special person of the kind one encounters a few times in life, which has been an excellent friend and to whom I also owe a lot.

My gratitude to my thesis committee, who took the heavy job of reading the whole thesis, and made a number of useful comments that improved the work in many ways: Jorge Olivos (again), Patricio Poblete (again) and Esko Ukkonen.

There are many other people which I am indebted to for this thesis. Many of them, for instance, sent me working versions of their algorithms, what made the tests a lot easier and, in some sense, more fair: William Chang, Alden Wright, Gene Myers, Erkki Sutinen, Tadao Takaoka, Jorma Tarhio, Robert Muth, Udi Manber and Archie Cobbs. Others have read and made suggestions to improve papers which later became part of this thesis, such as Gene Myers, Udi Manber, Erkki Sutinen, and of course a lot of anonymous conference and journal referees. Finally, some people have worked with us in papers related to this thesis, and although I have only included here my original work, they have worked very close to us and are responsible of many improvements: Erkki Sutinen, Jorma Tarhio, Nivio Ziviani, Marcio Drumond Araújo, Edleno de Moura and Mathieu Raffinot. In Chapter 8 I have borrowed some experimental figures from joint papers (thanks to Nivio and Marcio).

Last but not least, FONDECYT (Chile) grants 1-950622 and 1-960881, FONDEF (Chile) grant 96-1014, and CYTED VII.13 AMYRI Project, which partially supported this work, are gratefully acknowledged.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis focuses on the problem of *text retrieval allowing errors*, also called *approximate string matching*. The problem is to find a pattern in a text where the pattern and the text may have "errors". We first give some historical background and motivation for this problem, then define it precisely, and finally survey what are our contributions.

## 1.1 History and Motivation

The first references to this problem we could trace are from the sixties and seventies, where the problem appeared in a number of different areas. In those times, the main motivation for this kind of search came from the fields of computational biology, signal processing and text processing. We first consider the alternative fields and later concentrate on text.

DNA and protein sequences can be seen as long texts over specific alphabets (e.g. {`A`,`C`,`G`,`T`} in DNA). Those sequences represent the genetic code of living beings. Searching specific sequences over those texts appeared as a fundamental operation for problems such as looking for given features in DNA chains, or determining how different two genetic sequences were. This was modeled as searching for given "patterns" in a "text". However, exact searching was of no use for this application, since the patterns rarely matched the text exactly. The genetic sequences of two members of the same species are not identical, they are just very similar. Moreover, establishing how different two sequences were (e.g. to determine how far in the past did mice and men diverge) needed also such concept of "similarity", as well as an algorithm to compute it.

This gave a motivation to "search allowing errors". The errors were those operations that biologists knew were common to occur in genetic sequences. The "distance" between two sequences was defined as the minimum (i.e. more likely) sequence of operations to transform one into the other. With regard to likelihood, the operations were assigned a "cost", such that the more likely operations were cheaper. The goal was then to minimize the total cost.

As mentioned, another motivation came from the area of signal processing. One of the largest areas deals with speech recognition, where the general problem is to be able to determine, given an audio signal, a textual message which is being transmitted. Even simplified problems such as discerning a word from a small set of alternatives is complex, since parts of the the signal may be compressed in time, parts of the speech may not be pronounced, etc. A perfect match is practically impossible.

Another problem of this field is error correction. The physical transmission of signals is error-prone.

To ensure correct transmission over a physical channel, it is necessary to be able to recover the correct message after a possible modification (error) introduced during the transmission. The probability of such errors is obtained from the signal processing theory and used to assign a cost to those errors. In this case we may even not know what we are searching for, we just want a text which is correct (according to the error correcting code used) and closest to the received message. Although this area has not developed too much with respect to approximate searching, it has generated the most important measure of similarity, known as the Levenshtein (or edit) distance [Lev65, Lev66].

Computational biology has since then evolved and developed a lot, with a special push in the last years due to the projects that aim to the complete decoding of the DNA and its potential applications. Signal processing is also a very active area. Multimedia interfaces put a pressure on non-written communication. Strong error correcting codes are also sought given the current interest in wireless networks. Of course the problems that appear in these areas related to approximate searching are not only of the kind we have presented. For instance, one may look for an unknown pattern, of which only some properties are known. Even the simple problem of a known pattern is believed to be NP-complete under some distance functions (e.g. sorting by reversals [KS95, PW95]).

We refer the reader interested in more information on the birth of this area to [SK83]. In particular, good references for the applications of approximate pattern matching for computational biology are [Wat95, BSSU74, WL83, GK82, KG82], and for signal processing are [DM79, Lev65, Vin68, LS97].

Nowadays, a number of new applications for approximate string matching have appeared. The rapidly evolving field of multimedia databases needs algorithms to search a pattern in a physical signal (not only audio), where it is practically impossible to find it exactly. The number of applications for this problem grows every day. We have found solutions to the most diverse problems based on approximate string matching, for instance handwriting recognition, virus and intrusion detection, image compression, data mining, pattern recognition and automated video editing, to name a few. Many more applications are mentioned in [SK83]. Moreover, it is not necessary to go that far, since simple tools of everyday use such as the ability to find the difference between two files[1] make heavy use of it.

Especially interesting for this thesis is the case of text searching. The problem of correcting misspelled words in written text is rather old, perhaps the oldest potential application for approximate string matching. We could find references from the twenties [Mas27], and perhaps there are older ones. However, some time elapsed until it was realized [Nes86] that the ad-hoc models (e.g. those of Blair [Bla60], Damerau [Dam64] and the popular Soundex, described for instance in [Knu73, HD80]) were inferior to the approximate string matching approach.

Nowadays, the spectrum of problems in this area is much wider. The amount of textual information available worldwide is impressive for its size. The World-Wide-Web (or simply Web) contains more than one terabyte. Even specific text collection sizes are measured in gigabytes. Finding the relevant information on such mass of text becomes a more and more complex matter. Searching natural language text is not like searching exact information in, say, a relational database. One is normally interested in querying a text based on its *semantics*, which is not easy to extract from the content.

This problem opens the door to the vast field of Information Retrieval (IR, another long standing area in computer science), whose main goal is to find which text documents may be of interest to a user given his/her query. This is a rather heuristic discipline since it is based on maximizing vague concepts such as the user satisfaction. We refer the reader interested in IR to [FBY92, BYR98]. On

[1]The Unix "`diff`" command, for instance.

the other hand, we are interested in the algorithmic aspects of the problem. Most of the techniques used in IR to find "relevant" documents are based on text searching algorithms. We aim at obtaining such efficient search algorithms.

To make the problem even harder, those text collections are becoming more and more heterogeneous (different languages, for instance) and more error prone. For instance, text collections digitalized via optical character recognition (OCR) contain a non-negligible percentage of errors. The same happens with typing errors. Many text databases are so large and grow so fast that it is impossible to control their quality (in particular, there is no quality control in the Web). A word which is entered incorrectly in the database will never be retrieved unless the query makes the same mistake. A recent experiment has shown that close to 10% of the interesting documents relevant to a given query on the Web are not retrieved because of such errors [Rib97].

Notice that not only the text but also the pattern may have errors. This is typical, for instance, on cross-lingual scenarios where a foreign name is sought and is incorrectly spelled in the search pattern, or ancient texts which use outdated versions of the language. Finally, other text processing applications such as spelling checkers (which suggest "close" variants to the misspelled word) need algorithms to search for misspelled words. One may argue that spelling checkers will avoid in the future the presence of errors in the text, but apart from the fact that those spelling checkers need anyway from approximate searching tools, we know that it is difficult to perform automatic correction, as we see shortly in an example.

Many techniques are used to increase the probability of finding the relevant text portions. Apart from techniques which are typical from IR, such as the use of thesauri, stemming algorithms and others, it is important to have flexibility in the search algorithms themselves. A number of such new capabilities are grouped under the generic name of *extended patterns*, which generalize the basic exact search query. These generalizations range from disregarding the lower- or upper-case of letters to searching for regular expressions. Nowadays, there is virtually no text retrieval product that does not allow some kind of extended search facility.

Among all the alternatives offered under the name "extended patterns", the one which best copes with error-prone texts or patterns derives precisely from those well-known problems of approximate string matching. A simple adaptation has received a lot of attention. It is a cost model called *Levenshtein distance* or simply *edit distance*, denoted $ed()$ [Lev65]. This is defined as the minimum number of character insertions, deletions and replacements to make two strings equal. For instance, $ed($`"survey"`,`"surgery"`$) = 2$. See for instance [Nes86], which shows that simple versions of the edit distance function outperform in precision all the other known methods.

Let's illustrate our case with a simple example. Consider searching the word `"against"` in 1.2 Gb of the TREC-2 [Har95] collection, which is a standard set of texts used for comparing IR products. Our search allowing one insertion, deletion or substitution shows

```
aagainst[1]  abainst[1]   agaimst[1]    againist[2]  agains[7]
against      againsts[2]  againt[23]    agaist[5]    aganinst[1]
aganst[4]    aginst[13]   agsainst[1]   gagainst[1]  tgainst[1]
```

all of which, apart from `"against"` itself, are erroneous variants of `"against"`. We put in square brackets the number of times each of the erroneous variants appears. As an example against automatic error correction, the word `"agains"` was in some cases an erroneous variant of `"against"` and in others of `"again"`, and we had to read the paragraph carefully to select among the two. If we search allowing two errors, we obtain all the previous words and also

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| `advinst` | `again` | `againg` | `againns[1]` | `againto` | `againts[2]` |
| `agaisnt[7]` | `aganet` | `aganist[3]` | `agianst[1]` | `agins` | `aginse` |
| `ainst[1]` | `anainsa` | `gains` | `gainse` | `gaint` | `ragains` |

some of which are still erroneous variants of `"against"`. In particular four different alternatives of a letter transposition appears (that is, converting `"ab"` to `"ba"`). This is a typical typing error (in fact it may be better to consider it as an atomic error). Other variants are in fact different correct words, such as `"again"` and `"gains"`, and some others are incorrect variants of these different words, e.g. `"againg"` and `"gainse"`.[2]

Summing up the number of times our word is misspelled yields 78 occurrences (3 of which are in headlines and titles!), against 77,556 correctly written (close to 0.1%). We remark that this collection comes from published articles and newspapers, i.e. it is not a carelessly-written material.

We tried to do the same in the Web using Altavista (`http://www.altavista.com`). Since Altavista does not allow to perform an approximate search, we content ourselves with asking for all the incorrect variants previously found. The result is 22,610 pages with incorrect spellings (unfortunately we cannot check all them!), which represents 0.5% of the total number of pages where `"against"` was found.

In the above example we selected a very common word to illustrate the many misspelled variants that may appear. However, the result may be a little deceptive, since we are losing less than 1% of the matches by forgetting about approximate searching. But let's consider a word which is more difficult to spell: `"Levenshtein"`. Altavista found 192 pages for us. Now, we tried all the alternatives of deleting one letter and of exchanging one letter with the next one. The result is that 87 new pages appeared! (and we checked that they were relevant). That is, we were missing one third of the relevant pages. We remark that we could not try insertions or replacements because the number of exact searches to perform would be huge. This serves also as an illustration of how difficult is to perform approximate searching when there is no provision for it from the text retrieval tool. Moreover, it illustrates the fact that uncommon words have higher probability of being misspelled, and are precisely those uncommon words the most important for IR and most commonly queried.

## 1.2 The Problem

We are interested in approximate string matching in general, although our strongest emphasis is on text searching. Our problem can be stated as follows:

> Given a short pattern $P$ of length $m$, a long text $T$ of length $n$, and a maximal number of errors $k$, find all text positions $j$ such that a suffix of $T...j$ matches $P$ with at most $k$ errors (insertions, deletions or replacements).

Notice that we return the ending positions of matches. This decision is taken because the matched text portions may have different lengths. Equivalently we could report the beginnings of matches. It has been traditionally preferred to report ending positions because this simplifies the search algorithms.

The use of the Levenshtein distance instead of any other is a consequence of our focus on text searching applications. Many algorithms we propose, however, can be generalized to allow other

[2]We checked each of these candidates in its textual context to determine which word they corresponded to.

costs for the edit operations. This includes, in particular, other commonly used cost models such as the Hamming distance (i.e. allow only replacements, also called substitutions or mismatches), the Longest Common Subsequence distance (i.e. allow only insertions and deletions), the Episode Distance (i.e. allowing only insertions), or an extended edit distance that allows letter transpositions.

On the other hand, we will be interested in combining this type of search with the typical requirements of modern information retrieval, such as allowing errors in extended patterns, matching whole words, etc. Because of the huge text sizes involved, we will be interested in indexing techniques to ease this search. Finally, we will also be interested in different variants and extensions of this problem.

Being interested in typical text retrieval applications involves also being more interested in some areas of the parameters than in others (although some of our algorithms work better in other areas). There is no algorithm which is optimal in all cases. The parameters of interest are:

- The text size, which is assumed to be huge, in contrast to, e.g., many computational biology applications where it may be a few thousands of letters. This, for instance, discourages the use of memory-resident data structures which are many times the text size.
- The pattern length, which is assumed to be reasonably small, typically no longer than 30 letters and almost always shorter than 60 letters. This rules out in practice many algorithms which are good only for very long patterns. Those algorithms can be of use in computational biology applications, where it is common to have a pattern of a few hundreds of letters.
- The error level, which is reasonably low (say, $k/m \leq 1/2$, and normally $k/m \leq 1/3$). This is because, in text retrieval environments, allowing more errors will return a large portion of the text database, which makes the query useless for its low precision.
- The alphabet, which is not very small and not random. In contrast, the alphabet can be as small as four symbols in DNA, and much more random. Many algorithms rely on having a sufficiently large alphabet, and some try to address the problem of non-uniformity.

## 1.3 Overview of the Thesis

This thesis was developed between 1995 and 1998. Its aim is the development and analysis of novel algorithms to deal with the problem under various conditions, as well as a better understanding of the problem itself and its statistical behavior. Although our results apply to many different areas, we focus our attention on typical text searching for information retrieval applications.

As a whole, we believe that this work constitutes a valuable contribution to the development and understanding of the problem of approximate text searching.

We have divided the thesis in two parts: on-line and indexed searching. There is also some general material in the initial and final chapters.

### 1.3.1 General Part

This is composed of introductory chapters, conclusions, and some results which are applied in many places of the thesis.

- In Chapter 2 we explain all the basic concepts needed to read the thesis.

- In Chapter 3 we cover the related work and put our contributions in context.
- In Chapter 4 we present our first results. We obtain very tight bounds for the probability of a random pattern matching at a given position in random text with a given number of errors. This theoretical analysis is confirmed and tightened with extensive experimental validation. This result is not only essential to analyze our new algorithms, but it is also used to improve the analysis of previous algorithms.
- The final chapter of the thesis (Chapter 10) presents a perspective view and some insights on the future of the field.

### 1.3.2 On-line Searching

The first part of the thesis deals with on-line approximate string matching. This is the area where more progress has been made since the sixties, and a wealth of competitive algorithms exist. On-line searching is of interest not only because in many cases there is no time or space to preprocess the text, but also because it lies at the heart of most indexing techniques. We point out our major achievements in this part. This work has been published in [BYN96b, BYN96a, BYN97b, Nav97a, Nav97b, BYN98d, BYN98c, NBY98b, NBY98d, NR98b], and there are more submitted (this includes the results of Chapter 4).

- We develop in Chapter 5 a new algorithm for approximate string matching. This combines bit-parallelism with filtering techniques (to be explained later). As the use of bit-parallelism limits the length of the patterns to use, we develop a number of problem partitioning techniques that we combine in an optimal way (using also some techniques developed in Chapter 4). The final algorithm turns out to be the fastest known algorithm for many interesting ranges of parameters, including short patterns and low error levels. Moreover, it can be generalized to support many extended patterns. The design of this algorithm is inextricably mixed with its analysis, being the analysis (and hence the results of the previous point) an important part of its design.
- As a subproduct of the previous point, we rescue in Chapter 6 a previous algorithm which received little attention prior to this thesis. The algorithm is called "partitioning into exact search", since it is based on multipattern exact searching. With a clever implementation and the inclusion of more sophisticated subsidiary techniques (developed in Chapter 4), it becomes the fastest known algorithm for low and moderate error levels and pattern lengths. Moreover, using a different multipattern search technique, almost the same efficiency is obtained while being able to handle some extended patterns.
- Also in Chapter 6, we develop a generalized suffix automaton technique that allows improving the search time for many algorithms, and apply it to our bit-parallel algorithm. The result improves over the original algorithm for some ranges of parameters (especially in DNA searching).
- We improve (Chapter 6) the implementation and analyze two other algorithms which were partially developed prior to this thesis. The first one is a filter based on counting matching characters, for which we give the first theoretical average-case analysis. The second one is a partially built deterministic automaton to search the pattern allowing errors, for whose growth we give the first empirical analysis.

- We address in Chapter 7 the problem of multipattern approximate searching, which consists of having a number of patterns to search at the same time. This is of interest, for instance, in spelling applications (e.g. look for the whole dictionary in the text, finding the incorrect words and their closest correct spellings), in information retrieval systems (e.g. expand the search word using a thesaurus or synonyms and search all them allowing errors because the text has errors), in batch searching applications (e.g. perform all the accumulated queries at once), among others.

  The only previous work allowed searching thousands of patterns in parallel with just one error. We adapt our bit-parallel, partitioning into exact search, and counting algorithms, for the case of multiple patterns. The result is the fastest algorithms in all cases when either the number of patterns is not too large (say, less than 100) or more than one error is allowed. We analyze and experimentally compare all the algorithms.

- All our algorithms, as well as the best previous algorithms we are aware of, were implemented (or we obtained the implementations from their authors) and experimentally tested for many interesting ranges of the parameters of the problem.

We have also obtained new results for different variants of the approximate searching problem [Nav98a, Nav98b, BYN98c, NBY98a, NR98b, NR98c, MNZBY98b, MNZBY98a]. These variants include approximate searching on multi-dimensional text, on non-linear text (hypertext), on compressed text, and using the Hamming distance. Because these related results do not fit well with the overall content of this thesis, we have decided not to include them.

### 1.3.3 Indexed Searching

The second part addresses the problem of building indices (i.e. data structures on the text) that improve later searches of patterns allowing errors. This area was (and is) rather undeveloped. The problem of designing a good index for approximate searching is so difficult that some computer scientists have referred to the problem as the "Holy Grail" of this area. The existing indices are very large in comparison to the text (10 or 20 times the text size is a common figure), and give little or no guarantee on their performance. We call "sequence-retrieving" those general indices.

On the other hand, specialized indices for natural language have been more successful. They are basically inverted indices, making heavy use of the text vocabulary and being able to retrieve just whole words that match the pattern (e.g. if the error is the insertion of a space in the middle of the word, the index is not able to recover from it). We call "word-retrieving" those indices. We can subdivide them in those which know the exact text positions of all words ("full inverted indices") and those which reduce space requirements by dividing the text in blocks and point to the blocks instead of the exact positions ("block-addressing indices").

We have obtained new indexing and searching techniques and novel analytical results on some of them. This work has been published in [BYNST97, BYN97a, ANZ97, BYN97c, BYN98b, NBY98c, BYN98a], and there are others submitted. Our main achievements in this regard follow.

- We consider first full-inverted indices in Chapter 8. In this case, we prove that for most reasonable queries (i.e. those with reasonably high precision), the search time on those indices is sublinear in the text size, being close to the square root of the text size.

- In the same chapter, we prove analytically that word-retrieving block-addressing indices can be asymptotically sublinear in their size (with respect to the text) and in their time to answer approximate search queries. We find the expression for the block size that achieves this result and experimentally verify the theoretical results. This makes an excellent case for this kind of indices.

- Also in Chapter 8 we improve the vocabulary search of all word-retrieving indices. In previous work the vocabulary of the text is sequentially searched with an on-line algorithm. We give the vocabulary the structure of a metric space and compare different data structures to search in that space. The final result is a reduction of the vocabulary search time in up to 60%.

- We finish Chapter 8 by improving the existing block searching algorithms for word-retrieving block-addressing indices, making them up to five times faster. Those improvements find also application in text compression schemes for natural language texts. Those compression schemes allow searching a pattern permitting errors in the compressed text without decompressing it, which is an open problem in [ABF96].

- In Chapter 9 we propose and experimentally evaluate a new sequence-retrieving indexing scheme based on taking text samples. This is a variant of another index proposed prior to this thesis, and can be seen as an indexed version of the on-line algorithm called "partitioning into exact search". In comparison to the related previous index, the new one takes more space but it is more tolerant to errors. In comparison to most other indices, it takes much less space. This makes it a practical alternative for text searching.

- Also in Chapter 9 we adapt a previously existing sequence-retrieving indexing scheme. This index simulates the on-line text traversal on a data structure that eliminates the repetitions of the text. We use our bit-parallel algorithm as the on-line searcher. We also develop a new algorithm on the same data structure based on an adaptation of the pattern partitioning techniques, i.e. searching parts of the pattern with less errors and combining the results. We analyze the performance of the resulting index and find experimentally that it outperforms by far all the others. We propose for future work to simulate it on an index of text samples, to reduce its space requirements.

- We finish that chapter with a practical comparison among ours and all the other indexing proposals for retrieving sequences.

# Chapter 2

# Notation and Basic Concepts

We collect in this chapter all the important concepts and notations needed to read this thesis. There are no new results in this section. A basic understanding is assumed on design and analysis of algorithms and data structures, basic text algorithms, and formal languages. If this is not the case we refer the reader to good books in these subjects, such as [AHU74, CLR91, Knu73] (general), [GBY91, CR94] (for text algorithms) and [HU79] (for formal languages).

Table 2.1 summarizes the variables used along this thesis. A quick reminder of their meaning and values is included. A full explanation can be found throughout this chapter. We do not include the basic notation about strings and bit masks, which can also be found in this chapter.

We specify also some notation regarding time and space complexity. When we say that an algorithm is $O(x)$ time we refer to its worst case (although sometimes we say that explicitly). If the cost is on average, we say so explicitly. We also say sometimes that the algorithm is $O(x)$ cost, meaning time. When we refer to space complexity we say so explicitly. The average case analysis normally assumes a random text, where each character is selected uniformly and independently from the alphabet. The pattern is not normally assumed to be random (in some cases we explicitly state more specific assumptions on the randomness).

It is also convenient to define at this point the machine used for most of the experimental results we present along this work. This is the default machine, unless we explicitly say that another machine has been used. We use a Sun UltraSparc-1 of 167 MHz with 64 Mb of RAM and a 2 Gb local disk. The operating system is Solaris 2.5.1. This machine was not performing other heavy tasks when the experiments were run. We measured user times (i.e. CPU times), and in some cases system time (i.e. I/O time) or elapsed time. All the experiments were repeated enough times so as to ensure a given relative error with a given confidence interval. The confidence interval is typically 95%, while the relative error is 10% except when otherwise indicated.

We use two types of text. When we refer to "random text" we mean a text which is randomly generated in the sense that each character is uniformly and independently chosen over an alphabet of size $\sigma$. The search patterns are also generated using the same technique. When, on the other hand, we mention "English text", we refer to English literary text which is filtered so that all the letters are converted to lower-case and all the separators (except end of lines) to space. The patterns are chosen from the same text at random positions which start words no shorter than four letters. This mimics common Information Retrieval scenarios. However, we have obtained the same behavior on other languages such as Spanish or French. Moreover, the results (except for their higher variance) are very similar to those on random text on an alphabet size of $1/p$, where $p$ is

| Name | Meaning | Domain | Typical values |
|---|---|---|---|
| $\Sigma$ | Alphabet | Finite set | |
| $\sigma$ | Alphabet size | $\mathbb{N}, \geq 2$ | 4 or 20 to 64 |
| $ed()$ | Edit distance function | $\Sigma^* \times \Sigma^* \rightarrow \mathbb{N}$ | Levenshtein distance |
| $T$ | Text | $\Sigma^*$ | Natural language or arbitrary text |
| $n$ | Text length in letters or words | $\mathbb{N}, > 0$ | Megabytes or gigabytes |
| $P$ | Pattern to search | $\Sigma^*$ | Same source as text |
| $m$ | Pattern length in letters | $\mathbb{N}, > 0$ | Up to 30 or 60 |
| $k$ | Number of allowed errors | $\mathbb{N}, \in \{1..(m-1)\}$ | Up to $m/3$ or $m/2$ |
| $\alpha$ | Error level $\alpha = k/m$ | $\mathbb{R}, \in (0..1)$ | Up to 1/3 or 1/2 |
| $w$ | Bits in the computer word | $\mathbb{N}, > 0$ | 32 or 64 |
| $M$ | Available memory in bytes | $\mathbb{N}, > 0$ | Few megabytes |
| $V$ | Vocabulary size of the text | $\mathbb{N}, > 0$ | Thousands |
| $b$ | Block size of the indices | $\mathbb{N}, > 0$ | Few Kbytes |
| $\theta$ | Zipf's constant | $\mathbb{R}, \geq 1$ | 1.5 to 2.0 |
| $\beta$ | Heaps' constant | $\mathbb{R}, \in (0..1)$ | 0.4 to 0.6 |
| $q$ | Length of $q$-grams | $\mathbb{N}, > 0$ | 3 to 5 |

Table 2.1: Main variables used in this thesis.

the probability that two random characters of the English text match. The specific English text used is, except otherwise stated, a collection of writings of Benjamin Franklin. In other cases we use texts from the TREC-2 collection [Har95], which we simply call "TREC" from now on.

## 2.1 Definition of the Problem

In the Introduction we have defined the problem of approximate string matching as that of finding the text positions that match a pattern with up to $k$ errors. We give now a more formal definition.

In the discussion that follows, we use $s, x, y, z, v, w$ to represent arbitrary strings, and $a, b, c...$ to represent letters. Writing a sequence of strings and/or letters represents their concatenation. We assume that concepts such as prefix, suffix and substring are known. For any string $s \in \Sigma^*$ we denote its length as $|s|$. We also denote $s_i$ the $i$-th character of $s$, for integer $i \in \{1..|s|\}$. We denote $s_{i..j} = s_i s_{i+1} ... s_j$ (which is the empty string if $i > j$). The empty string is denoted as $\varepsilon$.

Let $\Sigma$ be an alphabet of size $|\Sigma| = \sigma$, which is finite[1].

Let $T \in \Sigma^*$ be a *text* of length $n = |T|$.

Let $P \in \Sigma^*$ be a *pattern* of length $m = |P|$.

Let $k \in \mathbb{N}$ be the maximum error allowed.

Let $d : \Sigma^* \times \Sigma^* \rightarrow \mathbb{N}$ be the *distance function*.

The problem is: given $T$, $P$, $k$ and $d()$, return the set of all the text positions $j$ such that there exists $i$ such that $d(T_{i..j}, P) \leq k$.

[1]However, many algorithms can be adapted to infinite alphabets with an extra $O(\log m)$ factor in their cost. This is because the pattern can have at most $m$ different letters and all the rest can be considered equal for our purposes. A table of size $\sigma$ would now be replaced by a search structure over at most $m + 1$ different letters.

In most cases, however, the distance functions have a more precise form:

> The distance $d(x, y)$ between two strings $x$ and $y$ is the cost of the minimum-cost sequence of *operations* that transform $x$ into $y$. The cost of a sequence of operations is the sum of the costs of the individual operations. The cost of an operation is considered a positive real number. If it is not possible to transform $x$ into $y$ we say that $d(x, y) = \infty$.

If such distance function turns out to be symmetric (i.e. $d(x, y) = d(y, x)$) and always $d(x, y) < \infty$, then $d()$ is a *metric*, i.e. it satisfies the following axioms:

- $\forall x,\ d(x, x) = 0$
- $\forall x \neq y,\ d(x, y) > 0$
- $\forall x, y,\ d(x, y)\ =\ d(y, x)$
- $\forall x, y, z,\ d(x, z)\ \leq\ d(x, y) + d(y, z)$

The last property is called "triangular inequality". In particular, in text searching applications the operations of most interest are:

– *Insertion*: insert a new letter $a$ into $x$. An insertion operation on the string $x = vw$ consists in adding a letter $a$, converting $x$ into $x' = vaw$.

– *Deletion*: delete a letter $a$ from $x$. A deletion operation on the string $x = vaw$ consists in removing a letter, converting $x$ into $x' = vw$.

– *Replacement or Substitution*: replace a letter $a$ in $x$. A replacement operation on the string $x = vaw$ consists in replacing a letter for another, converting $x$ into $x' = vbw$.

– *Transposition*: swap two adjacent letters $a$ and $b$ in $x$. A transposition operation on the string $x = vabw$ consists in swapping two adjacent letters, converting $x$ into $x' = vbaw$.

We are now in position to define the most commonly used distance functions (although there are many others).

– *Levenshtein or Edit distance* [Lev65]: allows insertions, deletions and replacements, all costing 1. This can be rephrased as "the minimum number of insertions, deletions and replacements to make two strings equal". In the literature this problem is in many cases called "string matching with $k$ differences". The distance is symmetric, and it holds $0 \leq d(x, y) \leq \max(|x|, |y|)$.

– *Hamming distance* [SK83]: allows only replacements, which cost 1. In the literature this problem is in many cases called "string matching with $k$ mismatches". The distance is symmetric, and it is finite whenever $|x| = |y|$. In this case it holds $0 \leq d(x, y) \leq |x|$.

– *Episode distance* [DFG$^+$97]: allows only insertions, which cost 1. In the literature this problem is in many cases called "episode matching", since it modelizes the case where a sequence of events is sought, where all them must occur within a short period. This distance is not symmetric, and it may not be possible to convert $x$ into $y$ in this case. Hence, it holds that $d(x, y)$ is either $|y| - |x|$ or $\infty$.

– *Longest Common Subsequence distance* [NW70, AG87]: allows only insertions and deletions, all costing 1. The name of this distance refers to the fact that it measures the length of the longest pairing of characters that can be made between both strings, so that the pairings respect the order of the letters. The longer the pairing, the smaller the distance. The distance is symmetric, and it holds $0 \leq d(x,y) \leq |x| + |y|$.

In all cases except the episode distance we have a symmetric distance, since there are complementary operations. In those cases one can think that the changes can be made over $x$ or $y$. Insertions on $x$ are the same as deletions in $y$ and vice versa, and replacements can be made in any of the two strings to match the other.

This thesis is most concerned with the Levenshtein distance [Lev65, Lev66] which we also call by its alternative name "edit distance" and denote $ed()$. This is the most interesting distance for text retrieval applications. Although transpositions are of interest (especially in case of typing errors), there are few algorithms to deal with them (see, e.g. [LW75, WB74, Ukk85a]). However, we will consider them at some points of this thesis (notice that a transposition can be simulated with an insertion plus a deletion, but the cost is different). We also will point out when our work can be extended to have different costs of the operations (which is of special interest in computational biology), including the extreme case of not allowing some operations. This includes the other distances mentioned.

Other variants of the problem will also be of interest. For instance, in natural language text we may be interested in finding text *words* which match the pattern with errors, instead of finding any text segment. Most algorithms are easily adapted to this requirement.

Finally, we point out some notations and conditions that hold for text retrieval applications.

– The problem is interesting only for $k > 0$, otherwise we are in the problem of exact string matching, which is very different from the subject of this thesis. Some of our algorithms, however, behave competitively when faced to this case.

– If the Hamming or edit distance are used, then the problem makes sense for $k < m$, since if we can perform $m$ operations we can make the pattern match at any text position by means of $m$ replacements.

– Under the Hamming or edit distance, we call $\alpha = k/m$ the *error level*, which given the above conditions satisfies $0 < \alpha < 1$. This value gives an idea of the "error ratio" allowed in the match (hence, $100\alpha$ is the percentage of error allowed).

– In text retrieval we are normally interested in obtaining the maximum efficiency for: very large $n$, reasonably small $m$ (typically $m \leq 30$, and almost always $m \leq 60$), reasonably low $\alpha$ (typically $\alpha \leq 1/3$ and almost always $\alpha < 1/2$), and reasonably large alphabets (typically larger than 20, although not necessarily random). However, some of our algorithms turn out to be efficient in other cases as well, which makes them of interest for other applications.

## 2.2 Dynamic Programming Algorithm

We present now the first algorithm to solve the problem. It has been rediscovered many times in the past, in different areas, e.g. [NW70, WF74, San72, Sel80] (there are more references in [Ukk85a], and a very good compendium of the area in [SK83]). Although the algorithm is not very efficient, it is one of the most flexible ones to adapt to different distance functions. We present the version that computes the edit distance. It should be easy for the reader to extrapolate to other cases.

We first show how to compute the edit distance between two strings $x$ and $y$. Later, we extend that algorithm to search a pattern in a text allowing errors.

The algorithm is based on dynamic programming. Imagine that we need to compute $ed(x,y)$. A matrix $C_{0..|x|,0..|y|}$ is filled, where $C_{i,j}$ represents the minimum number of operations needed to match $x_{1..i}$ to $y_{1..j}$. This is computed as follows[2]

$$\begin{aligned} C_{i,0} &= i \\ C_{0,j} &= j \\ C_{i,j} &= \text{if } (x_i = y_j) \text{ then } C_{i-1,j-1} \\ &\phantom{=} \text{ else } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}) \end{aligned}$$

where at the end $C_{|x|,|y|} = ed(x,y)$.

The rationale of the above formula is as follows. First, $C_{i,0}$ and $C_{0,j}$ represent the edit distance between a string of length $i$ or $j$ and the empty string. Clearly $i$ (respectively $j$) deletions are needed on the long string. For two non-empty strings of length $i$ and $j$, we assume inductively that all the edit distances between shorter strings have already been computed, and try to convert $x_{1..i}$ into $y_{1..j}$.

Consider the last characters $x_i$ and $y_j$. If they are equal, then we do not need to consider them and the conversion proceeds in the best way we can convert $x_{1..i-1}$ into $y_{1..j-1}$. On the other hand, if they are not equal, we must deal with them in some way. Following the three allowed operations, we can delete $x_i$ and convert in the best way $x_{1..i-1}$ into $y_{1..j}$, insert $y_j$ at the end of $x$ and convert in the best way $x_{1..i}$ into $y_{1..j-1}$, or replace $x_i$ by $y_j$ and convert in the best way $x_{1..i-1}$ into $y_{1..j-1}$. In all cases, the cost is one plus the cost for the rest of the process (already computed). Notice that the insertions in one string are equivalent to deletions in the other.

The dynamic programming algorithm must fill the matrix in such a way that the upper, left, and upper-left neighbors of a cell are computed prior to computing that cell. This is easily achieved by either a row-wise left-to-right traversal or a column-wise top-to-bottom traversal. Figure 2.1 illustrates this algorithm to compute $ed($"survey", "surgery"$)$.

Therefore, the algorithm is $O(|x||y|)$ time in the worst and average case. However, the space required is only $O(\min(|x|,|y|)$. This is because, in the case of a column-wise processing, only the previous column must be stored in order to compute the new one, and therefore we just keep one column and update it. We can process the matrix row-wise or column-wise so that the space requirement is minimized.

On the other hand, the sequences of operations performed to transform $x$ into $y$ can be easily recovered from the matrix, simply by proceeding from the cell $C_{|x|,|y|}$ to the cell $C_{0,0}$ following the path (i.e. sequence of operations) that matches the update formula (notice that multiple paths may exist). In this case, however, we need to store the complete matrix.

This matrix has some properties which can be easily proved by induction (see, e.g. [Ukk85a]) and which make it possible to design better algorithms. Notice, for instance, that the values of neighbor cells differ in at most one.

[2]This formulation of the problem has been selected because it is simpler to explain for us. A more classical formulation is $C_{i,j} = \min(C_{i-1,j}+1, C_{i,j-1}+1, C_{i-1,j-1}+\delta(x_i,y_j))$, where $\delta(x,y)$ is zero if $x=y$ and 1 otherwise. The formulation we used can be deduced from the more classical one by noticing that $C_{i-1,j-1} \leq 1 + \min(C_{i-1,j}, C_{i,j-1})$.

|  |  | s | u | r | g | e | r | y |
|---|---|---|---|---|---|---|---|---|
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| s | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| u | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 |
| r | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 |
| v | 4 | 3 | 2 | 1 | 1 | 2 | 3 | 4 |
| e | 5 | 4 | 3 | 2 | 2 | 1 | 2 | 3 |
| y | 6 | 5 | 4 | 3 | 3 | 2 | 2 | **2** |

Figure 2.1: The dynamic programming algorithm to compute the edit distance between "`survey`" and "`surgery`". The bold entry is the final result.

We show now how to adapt this algorithm to search a short pattern $P$ in a long text $T$. The algorithm is basically the same, with $x = P$ and $y = T$ (proceeding column-wise so that $O(m)$ space is required). The only difference is that we must allow that any text position is the potential start of a match. This is achieved by setting $C_{0,j} = 0$ for all $j \in 0..n$. That is, the empty pattern matches with zero errors at any text position (because it matches with a text suffix of length zero).

The algorithm then initializes its column $C_{0..m}$ with the values $C_i = i$, and processes the text character by character. At each new text character $T_j$, its column vector is updated to $C'_{0..m}$. The update formula is

$$\begin{aligned} C'_i &= \text{if } (P_i = T_j) \text{ then } C_{i-1} \\ &\quad\ \text{else } 1 + \min(C'_{i-1}, C_i, C_{i-1}) \end{aligned}$$

The search time of this algorithm is $O(mn)$ and its space requirement is $O(m)$. This is a sort of worst case in the analysis of all the algorithms that we consider later. Figure 2.2 exemplifies this algorithm applied to search the pattern "`survey`" in the text "`surgery`" (a very short text indeed) with at most $k = 2$ errors. In this case there are 3 occurrences.

|  |  | s | u | r | g | e | r | y |
|---|---|---|---|---|---|---|---|---|
|  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| s | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| u | 2 | 1 | 0 | 1 | 2 | 2 | 2 | 2 |
| r | 3 | 2 | 1 | 0 | 1 | 2 | 2 | 3 |
| v | 4 | 3 | 2 | 1 | 1 | 2 | 3 | 3 |
| e | 5 | 4 | 3 | 2 | 2 | 1 | 2 | 3 |
| y | 6 | 5 | 4 | 3 | 3 | **2** | **2** | **2** |

Figure 2.2: The dynamic programming algorithm search "`survey`" in the text "`surgery`" with two errors. Bold entries indicate matching positions.

It is easy to adapt this algorithm for the other distance functions mentioned. It is just a matter of not considering the three editing operations. In the case of the episode distance, we must set $C_{i,0} = \infty$.

Finally, we point out that although we have presented a column-wise algorithm to fill the matrix, many other works are based in alternative filling styles. For instance, we can fill it row-wise (we need $O(n)$ space, but it pays in some cases, as we see later), or even by diagonals or "secondary" diagonals. Figure 2.3 illustrates.

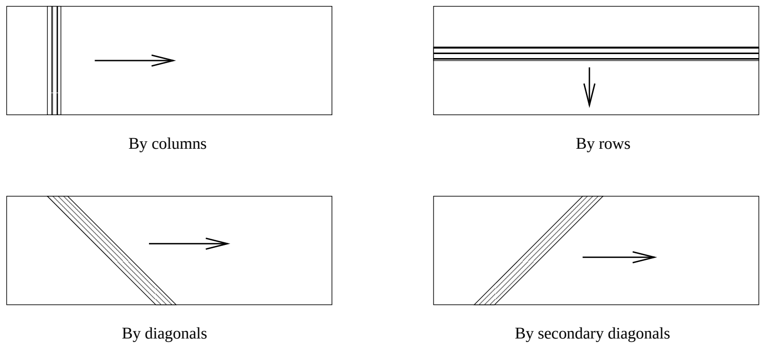


Figure 2.3: Different possible alternatives to fill the dynamic programming matrix.

## 2.3 A Graph Reformulation

A very powerful reformulation of the problem of computing edit distance converts the problem into a shortest-path problem in a graph [Ukk85a]. Given the two strings, the dynamic programming matrix can be seen as a graph where the nodes are the cells and the edges represent the operations. The weight of the edges correspond to the cost of the operations. There are also edges of zero cost when the characters match. The problem is then converted into finding the cheapest path from the node $[0, 0]$ to the node $[|x|, |y|]$ (and that path spells out the operations to perform). Figure 2.4 shows the graph for the example of Figure 2.1.

As it can be seen, diagonal arrows represent matches or replacements (with costs 0 or 1, respectively), horizontal arrows represent insertions in $x$ and vertical arrows represent deletions from $x$. Notice also that the shortest path (i.e. optimal sequence of edit operations) needs not be unique.

Although this graph has $(|x|+1)(|y|+1)$ nodes and therefore the minimum path algorithms are not efficient (they would take $O(|x||y|\log(|x||y|))$ using a classical algorithm), this reformulation has been extremely useful, especially because it is very flexible to adapt to other types of approximate searching problems. It is also the basis of useful algorithms and analytical results for the edit distance problem.

## 2.4 A Reformulation Based on Automata

An alternative and very useful way to consider the problem is to model the search with a non-deterministic automaton (NFA). This automaton (in its deterministic form) was first proposed in [Ukk85b], and first used in non-deterministic form (although implicitly) in [WM92a]. It is shown explicitly in [BY91, BY96, BYN96b, BYN98d].

Consider the NFA for $k = 2$ errors under edit distance shown in Figure 2.5. Each row denotes the number of errors seen. The first one 0, the second one 1, and so on. Every column represents matching the pattern up to a given position. At each iteration, a new text character is read and the

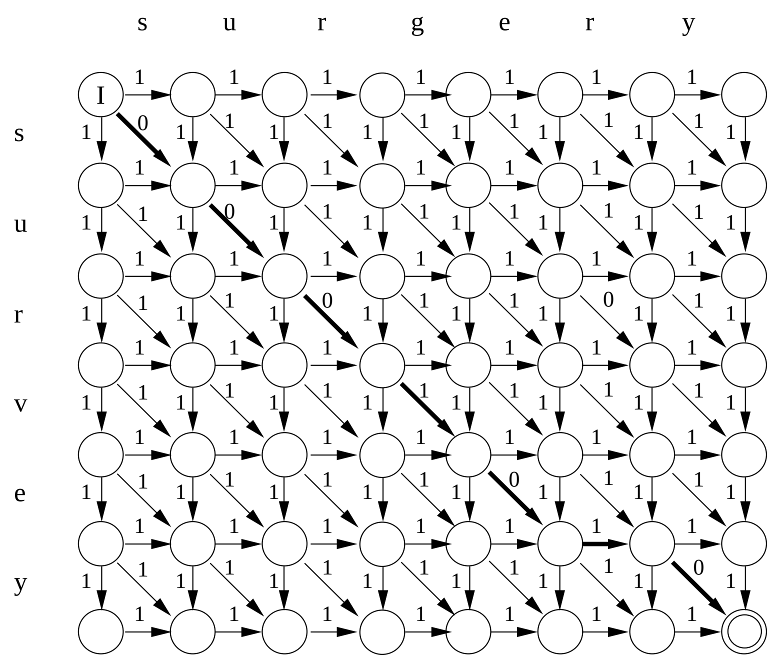


Figure 2.4: Graph reformulation of the previous edit distance problem. The cheapest path between the node “I” and the final node is shown in bold.

automaton changes its states. Horizontal arrows represent matching a character, vertical arrows represent insertions into the pattern, solid diagonal arrows represent replacements, and dashed diagonal arrows represent deletions in the pattern (they are $\varepsilon$-transitions). The automaton accepts a text position as the end of a match with $k$ errors whenever the rightmost state of the $(k+1)$-th row is active.

It is not hard to see that once a state in the automaton is active, all the states of the same column and higher rows are active too. Moreover, at a given text character, if we collect the smallest active rows at each column, we obtain the vertical vector of the dynamic programming algorithm [BY96] (compare to Figure 2.2).

The other types of distances are obtaining by deleting some arrows of the automaton. Different integer costs for the operations are also modeled by changing the arrows. For instance, if insertions cost 2 instead of 1, we make the vertical arrows to move from rows $i$ to rows $i+2$.

This automaton can be simply made deterministic to obtain $O(n)$ worst case search time. However, as we see later, the main problem becomes the construction of the DFA (deterministic finite automaton). An alternative solution is based on simulating the NFA instead of converting it into deterministic.

## 2.5 Filtering Algorithms

Many algorithms that we will consider and design are based in the concept of *filtering*, and therefore it is useful to define it here.

The concept of filtering is based on the fact that in many cases, the text positions matching a pattern with errors contain some parts of the pattern unaltered. For instance, if `"survey"` is found

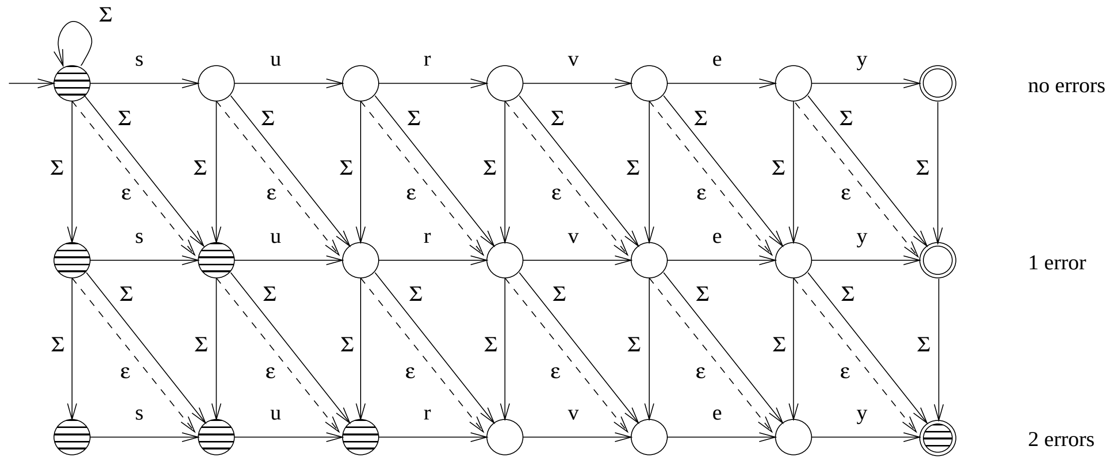


Figure 2.5: An NFA for approximate string matching of the pattern "survey" with two errors. The shaded states are those active after reading the text "surgery".

in a text position with one error under the edit distance, then either "sur" or "vey" must appear unaltered in the match. This is because a single edit operation cannot alter both halves of the pattern.

Most filtering algorithms take advantage of this fact by searching pieces of the pattern without errors. Since the exact searching algorithms can be much faster than approximate searching ones, filtering algorithms can be very competitive (in fact, they dominate on a large range of parameters).

It is important to notice that a filtering algorithm is normally unable to discover the matching text positions by itself. Rather, it is used to discard (hopefully large) areas of the text which cannot contain a match. For instance, in our example, it is necessary that either "sur" or "vey" appears in an approximate occurrence, but it is not sufficient. Any filtering algorithm must be coupled with a process that verifies all those text positions that could not be discarded by the filter.

Virtually any non-filtering algorithm can be used for this verification, and in many cases the developers of a filtering algorithm do not care in looking for the best verification algorithm, they just use the dynamic programming algorithm. It is important to understand that that selection is normally independent. Nonetheless, the verification algorithm must behave well on short texts, because it can be started in many different text positions to work on small text areas. By careful programming it is almost always possible to keep the worst-case behavior of the verifying algorithm (i.e. avoid verifying overlapping areas).

Finally, the performance of filtering algorithms is very sensitive to the error level $\alpha$. Most filters work very well on low error levels and very bad in other cases. This is related with the amount of text that the filter is able to discard. When evaluating filtering algorithms, it is important not only to consider their time efficiency but also their tolerance to errors.

A term normally used when referring to filters is "sublinearity". It is said that a filter is sublinear when it does not inspect all the characters of the text (like the Boyer-Moore [BM77] algorithms for exact searching, which can be at best $O(n/m)$). However, no on-line algorithm can be truly sublinear, i.e. $o(n)$. This is only achievable with indexing algorithms.

## 2.6 Bit-Parallelism

This is another technique of common use in string matching [BY92]. It was first proposed in [BY89, BYG92]. The technique consists in taking advantage of the intrinsic parallelism of the bit operations inside a computer word. By using cleverly this fact, the number of operations that an algorithm performs can be cut down by a factor of at most $w$, where $w$ is the number of bits in the computer word. Since in current architectures $w$ is 32 or 64, the speedup is very significant in practice (and improves with technological progress). In order to relate the behavior of bit-parallel algorithms to other works, it is normally assumed that $w = \Theta(\log n)$, as dictated by the RAM model of computation. We prefer, however, to keep $w$ as an independent value. We introduce now some notation we use for bit-parallel algorithms.

– The length of the computer word (in bits) is $w$.

– We denote as $b_\ell ... b_1$ the bits of a mask of length $\ell$. This mask is stored somewhere inside the computer word. Since the length $w$ of the computer word is fixed, we are hiding the details on where we store the $\ell$ bits inside it. We give such details when they are relevant.

– We use exponentiation to denote bit repetition (e.g. $0^3 1 = 0001$).

– We use C-like syntax for operations on the bits of computer words: "|" is the bitwise-or, "&" is the bitwise-and, "^" is the bitwise-xor and "~" complements all the bits. The shift-left operation, "<<", moves the bits to the left and enters zeros from the right, i.e. $b_m b_{m-1} ... b_2 b_1 << r \; = \; b_{m-r} ... b_2 b_1 0^r$. The shift-right, ">>" moves the bits in the other direction. Finally, we can perform arithmetic operations on the bits, such as addition and subtraction, which operates the bits as if they formed a number. For instance, $b_\ell ... b_x 10000 - 1 = b_\ell ... b_x 01111$.

We explain now the first bit-parallel algorithm, since it is the basis of much of which follows in this work. The algorithm searches a pattern in a text (without errors) by parallelizing the operation of a non-deterministic finite automaton that looks for the pattern. Figure 2.6 illustrates this automaton (from this point on, we use a pattern with more interesting combinatorial properties than "`survey`").

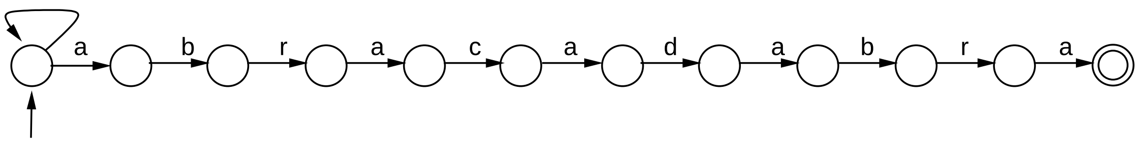


Figure 2.6: Nondeterministic automaton that searches "**`abracadabra`**" exactly.

This automaton has $m+1$ states, and can be simulated in its non-deterministic form in $O(mn)$ time. The Shift-Or algorithm achieves $O(mn/w)$ worst-case time (i.e. optimal speedup). Notice that if we convert the non-deterministic automaton to a deterministic one to have $O(n)$ search time, we get an improved version of the KMP algorithm [KMP77, BY96, Sim94, Han93]. However, for example, KMP is twice as slow for $m \leq w$.

The algorithm first builds a table $B$ which for each character $c$ stores a bit mask $B[c] = b_m ... b_1$. The mask in $B[c]$ has the bit $b_i$ in zero if and only if $P_i = c$. The state of the search is kept in a machine word $D = d_m ... d_1$, where $d_i$ is zero whenever $P_{1..i}$ matches the end of the text read up to now (i.e. the state numbered $i$ in Figure 2.6 is active). Therefore, a match is reported whenever $d_m$ is zero.

$D$ is set to all ones originally, and for each new text character $T_j$, $D$ is updated using the formula

$$D' \leftarrow (D << 1) \mid B[T_j]$$

The formula is correct because the $i$-th bit is zero if and only if the $(i-1)$-th bit was zero for the previous text character and the new text character matches the pattern at position $i$. In other words, $T_{j-i+1..j} = P_{1..i}$ if and only if $T_{j-i+1..j-1} = P_{1..i-1}$ and $T_j = P_i$. Again, it is possible to relate this formula to the movement that occurs in the non-deterministic automaton for each new text character: each state gets the value of the previous state, but this happens only if the text character matches the corresponding arrow.

For patterns longer than the computer word (i.e. $m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation (not all them are active all the time). The algorithm is $O(n)$ on average and the preprocessing is $O(m+\sigma)$ time and $O(\sigma)$ space.

It is easy to extend Shift-Or to handle *classes of characters*. In this extension, each position in the pattern matches with a set of characters rather than with a single character. The classical string matching algorithms are not so easily extended. In Shift-Or, it is enough to set the $i$-th bit of $B[c]$ for every $c \in P_i$ ($P_i$ is a set now). For instance, to search for `"survey"` in case-insensitive form, we just set the first bit of $B$[`"s"`] and of $B$[`"S"`], and the same with the rest. It can also search for multiple patterns (where the complexity is the same as before if we consider that $m$ is the total length of all the patterns). Shift-Or was later enhanced [WM92a] to support a larger set of extended patterns and even regular expressions.

Many on-line text algorithms can be seen as implementations of clever automata (classically, in their deterministic form). Bit-parallelism has since its invention became a general way to simulate simple non-deterministic automata instead of converting them to deterministic. It has the advantage of being much simpler, in many cases faster (since it makes better usage of the registers of the computer word), and easier to extend to handle complex patterns than its classical counterparts. Its main disadvantage is the limitations it imposes with regards to the size of the computer word. In many cases its adaptations to cope with longer patterns are not so efficient.

## 2.7 Suffix Trees and DAWGs

Suffix trees [Wei73, Knu73, AG85] are widely used data structures for text processing [Apo85]. Any position $i$ in a string $S$ defines automatically a *suffix* of $S$, namely $S_{i...}$.

In essence, a suffix tree is a trie data structure built over all the suffixes of $S$. At the leaf nodes the pointers to the suffixes are stored. Each leaf represents a suffix and each internal node represents a unique substring of $S$. Every substring of $S$ can be found by traversing a path from the root. Each node representing the substring $ax$ has a *suffix link* that leads to the node representing substring $x$.

To improve space utilization, this trie is compacted into a Patricia tree [Mor68]. This involves compressing unary paths. At the nodes which root a compressed path, an indication of how many characters to skip is stored. Once unary paths are not present the tree has $O(n)$ nodes instead of the worst-case $O(n^2)$ of the trie (see Figure 2.7).

This structure can be built in time $O(n)$, where $n = |S|$ [McC76, Ukk95]. It is a very useful structure to solve many problems [AG85, Apo85].

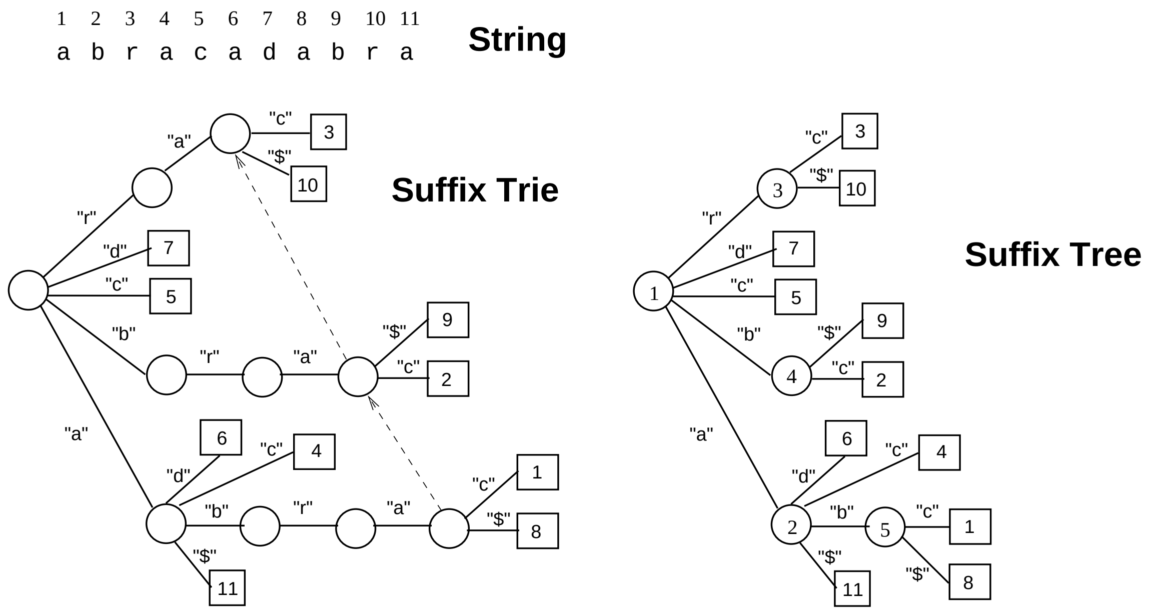


Figure 2.7: The suffix trie and suffix tree for a sample string. The “\$” is a special marker to denote the end of the text. Two suffix links are exemplified in the trie: from `"abra"` to `"bra"` and then to `"ra"`. The internal nodes of the suffix tree show the character position to inspect in the string.

To search a simple pattern in the suffix tree, we just enter into the trie driven by the letters of the pattern. This algorithm has been adapted to approximate searching by entering into *all* subtrees (since the matched text is not equal to the search pattern) until $k$ such errors are performed [Ukk93].

A DAWG (deterministic acyclic word graph) [Cro86, BBH$^+$85] built on a string $S$ is a deterministic automaton able to recognize all the substrings of $S$. As each node in the suffix tree corresponds to a substring, the DAWG is no more than the suffix tree augmented with failure links for the letters not present in the tree. Since final nodes are not distinguished, the DAWG is smaller. DAWGs have similar applications to those of suffix trees, and also use $O(n)$ space and construction time. Figure 2.8 illustrates.

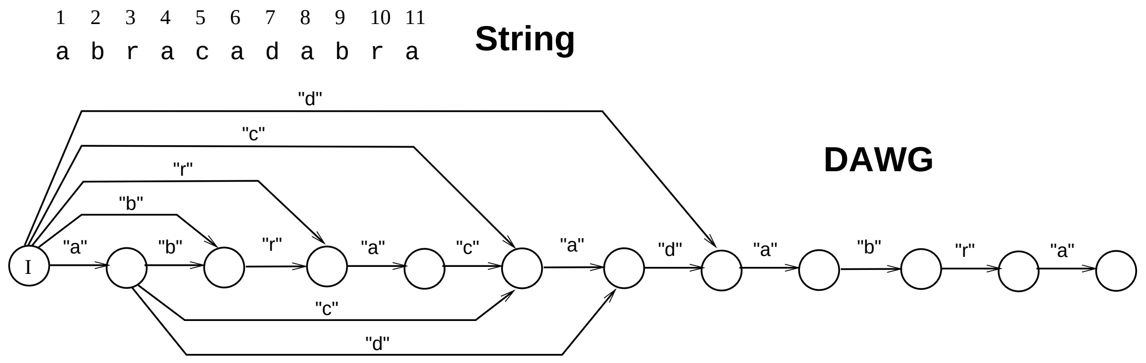


Figure 2.8: The DAWG or the suffix automaton for the sample string. If all the states are final, it is a DAWG. If only the rightmost state is final then it is a suffix automaton.

## 2.8 Suffix Automata

Although we make use of many different algorithms for exact string matching as building tools for our approximate search algorithms, we have decided to cover Shift-Or and this one because they are recent and not so widely known.

The Backward DAWG matching (BDM) algorithm [CCG+94, CR94] is based on a suffix automaton. A *suffix automaton* on a pattern $P$ is an automaton that recognizes all the suffixes of $P$. The non-deterministic version of this automaton has a very regular structure and is shown in Figure 2.9 (the deterministic version can be seen in Figure 2.8).

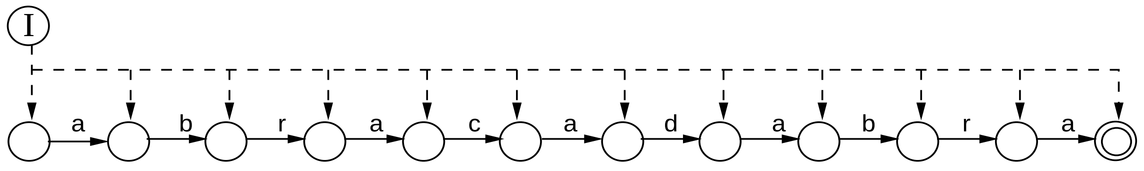


Figure 2.9: A non-deterministic suffix automaton to search any suffix of "abracadabra". Dashed lines represent $\varepsilon$-transitions (i.e. they occur without consuming any input). I is the initial state of the automaton.

The BDM algorithm converts this automaton to deterministic. The size and construction time of this automaton is $O(m)$. This is basically the preprocessing effort of the algorithm. Each path from the initial node to any internal node represents a substring of the pattern. The final nodes represent pattern suffixes.

To search a pattern $P$, the suffix automaton of $P^r$ (the reversed pattern) is built. The algorithm slides a window over the text and allows shifting the window without inspecting all the text characters. It searches backwards inside the text window for a substring of the pattern $P$ using the suffix automaton. Each time a terminal state is reached before reaching the beginning of the window, the position inside the window is remembered. This corresponds to finding a *prefix* of the pattern equal to a suffix of the window (since the reverse suffixes of $P^r$ are the prefixes of $P$). The last prefix recognized backwards is the *longest* prefix of $P$ in the window. A match is found if the complete window is read, while the check is abandoned when there is no transition to follow in the automaton. In either case, the window is shifted to align with the longest prefix recognized.

In [NR98a] we presented a bit-parallel version of BDM, called BNDM (the "N" stands for "non-deterministic"), based on simulating the non-deterministic automaton instead of converting it to deterministic. This version turned out to be the fastest algorithm to search exact patterns in all cases, except for very short (2-4 letters) or very long (100-150 letters) patterns. Moreover, it can handle some extended patterns easily and efficiently. Although this algorithm is not part of this thesis, we extend it in this work to handle errors.

## 2.9 Natural Language and Its Statistics

An important part of this work is oriented toward *natural language*. We explain in this section what is understood by this term and give some of the main accepted rules that drive its statistical behavior.

We understand by *natural language* text the transcription of a language used by humans for verbal communication, or which follows the same syntactic and semantic rules of such a language. This is in contrast to genetic sequences, compressed bit-streams, machine language, etc. Our definition encompasses all texts such as literary, technical, formal, colloquial, journalistic, e-mail, news, and many others, provided the language they are based on is used by humans to communicate (once the exceptions stated above are made, since two computer scientists could communicate via a program code).

Natural language and the mechanisms to extract information from it is the main subject of study of Information Retrieval (IR). A number of laws which rule the statistical behavior of natural language texts are widely accepted. Although we use them in many cases to analyze the average behavior of our algorithms and data structures (especially when we consider indices), it is important to realize that all those laws are *heuristic* and *approximate.* As we have found, some fit much better the reality than others. We present now the laws that we use in this work, some of which are fundamental for our results. All these laws have been experimentally confirmed in this thesis.

First of all, we assume that the natural language texts are divided in *words.* A *word* is a sequence of characters which forms a compact element. A more technical definition divides the alphabet $\Sigma$ in two sets, "letters" and "separators". Words are contiguous sequences of letters surrounded by separators (more complex definitions are possible). For instance, we could define that the letters are {`"a".."z"`, `"A".."Z"`, `"0".."9"`} and the rest are separators.

We assume that the words are relatively short, and that their length is distributed basically independently of their positions in the text. This assumption is very reasonable. In most cases of indexed search we assume that the text has $n$ words instead of $n$ characters. However, since the length of words is bounded by a constant, we still have a text of size $\Theta(n)$.

The *vocabulary* of the text is defined as the set of distinct words present in it, and its size is denoted as $V$. This definition may involve a previous step of mapping of characters and words. For instance, we may map all characters to lower-case, so that the words `"survey"` and `"Survey"` are considered equivalent. Some more complex mappings are commonly carried on IR systems. For instance, a word may be mapped to a *synonym* (i.e. a syntactically different word with the same meaning), so that if a user searches for `"only meaning"` it can also find `"sole meaning"`. Another technique, called *stemming*, eliminates a suffix of the words, so that if a user searches for `"clone"` it can also find `"cloning"`. All those transformations are assumed to be performed prior to the application of our algorithms and we do not rely on them.

We also assume the *Heaps' Law* [Hea78]. This is a very precise law ruling the growth of the vocabulary in natural language texts. It states that the vocabulary of a text of $n$ words is of size $V = Kn^\beta = O(n^\beta)$, where $K$ and $\beta$ depend on the particular text. $K$ is normally between 10 and 100, and $\beta$ is between 0 and 1 (not included). Some recent experiments [ANZ97, BYN97a] show that the most common values for $\beta$ are between 0.4 and 0.6. Hence, the vocabulary of a text grows sublinearly with the text size, in a proportion close to its square root.

A first inaccuracy appears immediately. Supposedly, the set of different words of a language is fixed by a constant (e.g. the number of different English words is finite). However, the limit is so high that it is much more accurate to assume that the size of the vocabulary is $O(n^\beta)$ instead of $O(1)$, although the number should stabilize for huge enough texts. On the other hand, many authors argue that the number keeps growing anyway because of the errors that appear in the text.

Another inconsistency is that, as the text grows, the number of different words will grow too, and therefore the number of letters to represent all the different words will be $O(\log(n^\beta)) = O(\log n)$. Therefore, longer and longer words should appear as the text grows. The average length could be kept constant if shorter words are common enough (which is the case). In practice, this effect is not noticeable and we can assume an invariable length, independent of the text size.

A much more inexact law is *Zipf's Law* [Zip49, GBY91], which rules the distribution of the frequencies (i.e. number of occurrences) of the words. The rule states that the frequency of the $i$-th most frequent word is $1/i^\theta$ times that of the most frequent word. This implies that in a text of $n$ words with a vocabulary of $V$ words, the $i$-th most frequent word appears $n/(i^\theta H_V(\theta))$ times, where

$$H_V(\theta) = \sum_{i=1}^{V} \frac{1}{i^\theta}$$

so that the sum of all frequencies is $n$. The value of $\theta$ depends on the text. In the most primitive formulation (which is easier to handle mathematically) it is assumed $\theta = 1$, and therefore $H_V(\theta) = O(\log V)$. However, this simplified version is very inexact, and the case $\theta > 1$ (more precisely, between 1.5 and 2.0) fits better the real data [ANZ97]. This case is very different, since the distribution is much more skewed, and $H_V(\theta) = O(1)$.

There have been attempts to correct the inaccuracies of Zipf's Law. One attempt is the Mandelbrot distribution [Man52], which states that the frequency of the $i$-th word is $n\mu/(c+i)^\theta$, for some constants $c$ and $\mu$. We do not use this distribution in this work because its asymptotical effect is negligible and it is much harder to deal with mathematically. It is interesting to notice that Zipf-like distributions can be derived from simpler models, as shown in [MNF57]: if we assume a word generation process where the space appears with a fixed probability $p$ and the other letters appear uniformly, then a Mandelbrot distribution in the generated words is observed. Moreover, shorter words are more frequent, as it happens in natural language.

The fact that the distribution of words is very skewed (i.e. there are a few hundreds of words which take up 50% of the text) suggest a concept which is also of much use in IR: *stopwords* [MNF58]. A *stopword* is a word which does not carry meaning in natural language and therefore can be ignored (i.e. made not searchable), such as `"a"`, `"the"`, `"by"`, etc. Fortunately, the most frequent words are stopwords, and therefore half of the words appearing in a text need not be considered. This allows, for instance, significantly reducing the space overhead of indices for natural language texts.

It is interesting to observe that if, instead of taking text words, we take $n$-grams, no Zipf-like distribution is observed. Moreover, no good model is known for this case [BCW90, chapter 4].

We point out now other assumptions we make. We assume that user queries distribute uniformly in the vocabulary, i.e. every word in the vocabulary can be searched with the same probability. This is not true in practice, since unfrequent words are searched with higher probability. On the other hand, approximate searching makes this distribution more uniform, since unfrequent words may match match with $k$ errors with other words, with little relation to the frequencies of the matched words. In general, however, the assumption of uniform distribution in the vocabulary is pessimistic.

Finally, the words are assumed to be uniformly distributed in the text. Although widely accepted, this rule may not be true in practice, since words tend to appear repeated in small areas of the text. Uniform distribution in the text is another pessimistic assumption we make, since more text blocks match a query when the distribution is uniform.

We end this section with a note of interest. There are many human languages where this clear separation of a text in words does not exist. This is the case of *agglutinating* languages, such as Finnish or German. In those languages, a sequence of letters is in fact the concatenation of a number of short components which carry the meaning. In many cases their words are equivalent to our phrases. All the statistical behavior exposed above will probably be untrue when applied to those languages. In that case, there are two main alternatives: ($a$) consider those texts as "non-natural languages" (in our sense, of course) and accept that the only schemes applicable to them are those for general texts; or (*b*) split those words into "sub-words" which really carry the meaning we assign to words in languages such as English, Spanish or French, for instance. This second alternative is very attractive, although it relies on some semantic understanding of the language. We are not aware of the existence or not of such semantic tools to split words in agglutinating languages.

## 2.10 Inverted Files or Inverted Indices

The *inverted file* or *inverted index* [Knu73, FBY92, WMB94] is a very old data structure to index natural language texts. *Indexing* means building a persistent data structure on texts to speed up the search. Since building an index takes much more than an on-line search, it is worthwhile to build such an index when the number of queries is much higher than the number of modifications to the text collection.

The inverted file structure is composed of two elements: the *vocabulary* and the *occurrences*. The vocabulary is the set of all different words in the text. For each such word a list of all the text positions where the word appears is stored. The set of all those lists is called the "occurrences" or "posting file" (Figure 2.10 shows an example).

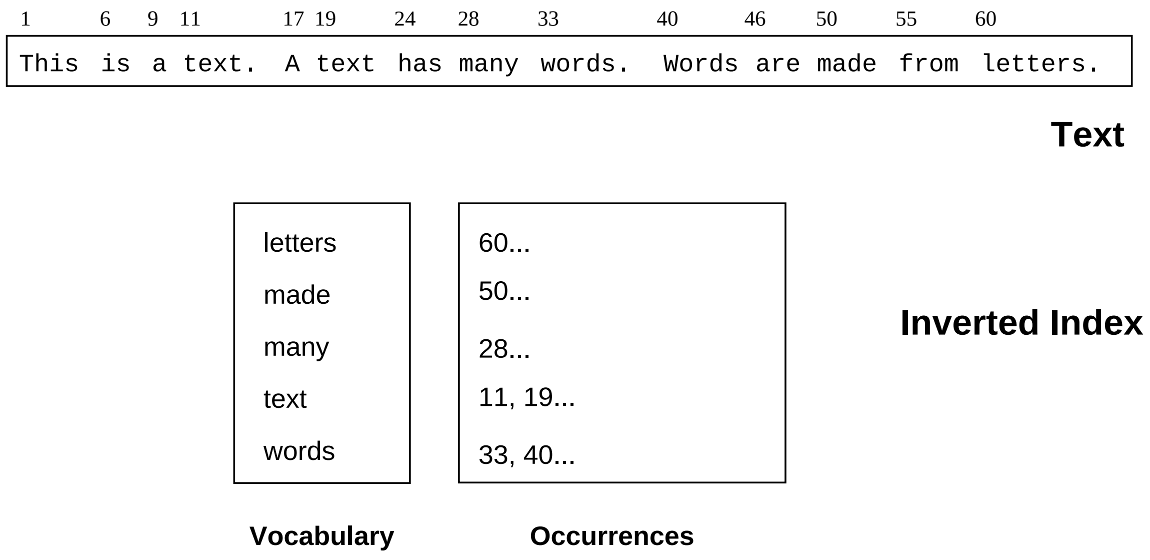


Figure 2.10: A sample text and an inverted index built on it. The words are converted to lower-case and stopwords are not indexed. The occurrences point to character positions in the text.

The space required for the vocabulary is rather small, thanks to the Heaps' Law. For instance, 1 Gb of the TREC collection [Har95] has a vocabulary whose size is only 5 Mb. The occurrences demand much more space. Since each word appearing in the text is referenced once in that structure, the extra space is $\Theta(n)$. Even omitting stopwords (which is the general case) in practice the space overhead of the occurrences is between 15% and 40% of the text size.

To reduce space requirements, a technique called *block addressing* is used. The first index of this kind was *Glimpse* [MW94]. The text is divided in blocks, and the occurrences point to the blocks where the word appears (instead of the exact positions). The classical indices which point to the exact occurrences are called “full inverted indices”. By using block addressing not only the pointers can be smaller because there are less blocks than positions, but also all the occurrences of a word inside a single block are collapsed to one reference (see Figure 2.11), profiting from context locality inherent to natural language. Indices of only 5% overhead over the text size are obtained with this technique. The price to pay is that, if the exact occurrence positions are required (for instance to solve a phrase query) then an on-line search over the classifying blocks has to be performed. For instance, block addressing indices with 256 blocks stop working well with texts of 200 Mb of size.

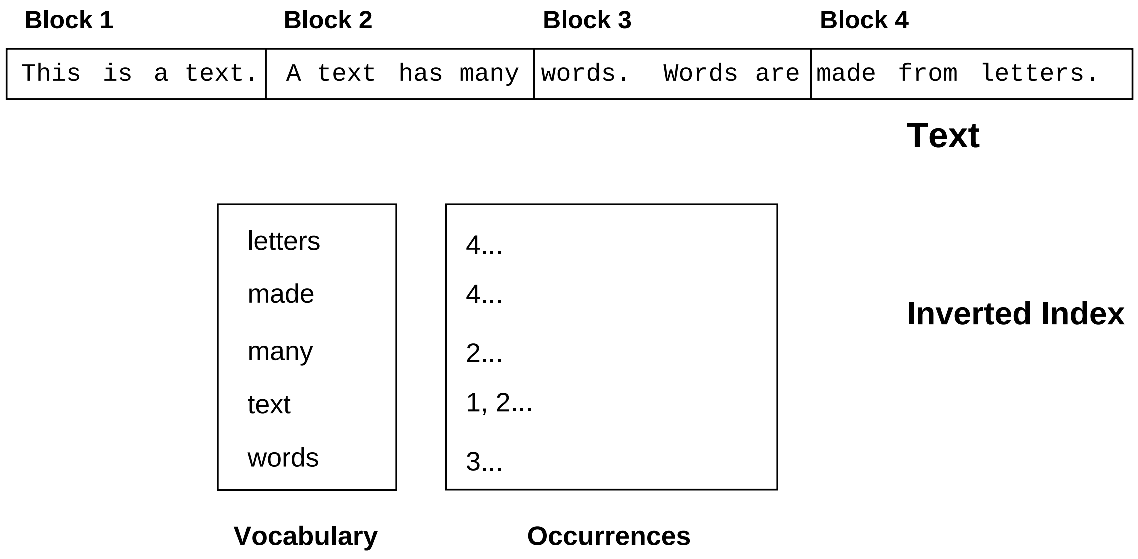


Figure 2.11: The sample text split in four blocks, and an inverted index using block addressing built on it. The occurrences denote block numbers. Both occurrences of `"words"` collapsed into one.

The blocks can be of fixed size (imposing a logical block structure over the text database) or they can be defined using the natural division of the text collection into files, documents, Web pages or others. The division into blocks of fixed size improves efficiency at retrieval time (i.e. the more variance in the block sizes, the more amount of text sequentially traversed on average).

On the other hand, the division using natural cuts may eliminate the need of the on-line traversal. For example, if one block per document is used and the exact match positions are not required, there is no need to traverse the text for simple queries, since it is enough to know which documents to report. On the other hand, if many documents are packed into a single block, the block has to be traversed to determine which documents to retrieve.

It is important to notice that in order to use block addressing, the text must be readily available at search time. This is not the case of remote text (as in Web search engines), or if the text is in a CD-ROM that has to be mounted, for instance.

Glimpse pioneered a second technique which we use many times in this work. It is based on a *sequential* vocabulary search. Traditionally, the queries are searched in the vocabulary by means of a data structure built on it, such as a hash table, B-tree or other. However, a sequential search is useful when we are interested in more complex types of searches such as extended patterns or approximate string matching.

Imagine that a user is interested in a variant of the problem where we have to report *whole words* which match a query word with $k$ errors or less. For instance, if the user searches `"shallow"`, we do not want it to be reported with one error in the text `"hash allows"` or with zero errors in `"shallowing"`. The indices that solve this variant of the problem are called in this work *word-retrieving* indices, as opposed to the general *sequence-retrieving* indices which solve the original problem.

Since the vocabulary contains all the different words that appear in the text, we can search just the vocabulary to find the required matches (this is because the context of the words does not affect the search). Since the vocabulary is much smaller than the text, a sequential search on the vocabulary is a valuable technique to deal with a restricted case of a very complex problem. Once all the words that match in the vocabulary are identified, we retrieve the documents where any of them appears.

In some cases, an inverted index is used not on the words of the text but on *q-grams* of the text. A $q$-gram is an arbitrary substring of length $q$. When the text does not have a structure of words, or when we do not want to use such structure, the text can be arbitrarily split in $q$-grams, which act as words in the sense that we store all the different $q$-grams as the vocabulary and all their text positions as the occurrences. As we see later, many alternative exists, but basically the $q$-grams are taken at fixed intervals of length $h$ (i.e. $T_{h..h+q-1}, T_{2h..2h+q-1}, ...$). The selected $q$-grams may or may not overlap in the text. The values of $q$ and $h$ are decisive in the size of the resulting index, since the number of occurrences are $O(n/h)$ and the vocabulary grows exponentially with $q$.

## 2.11 Suffix Arrays

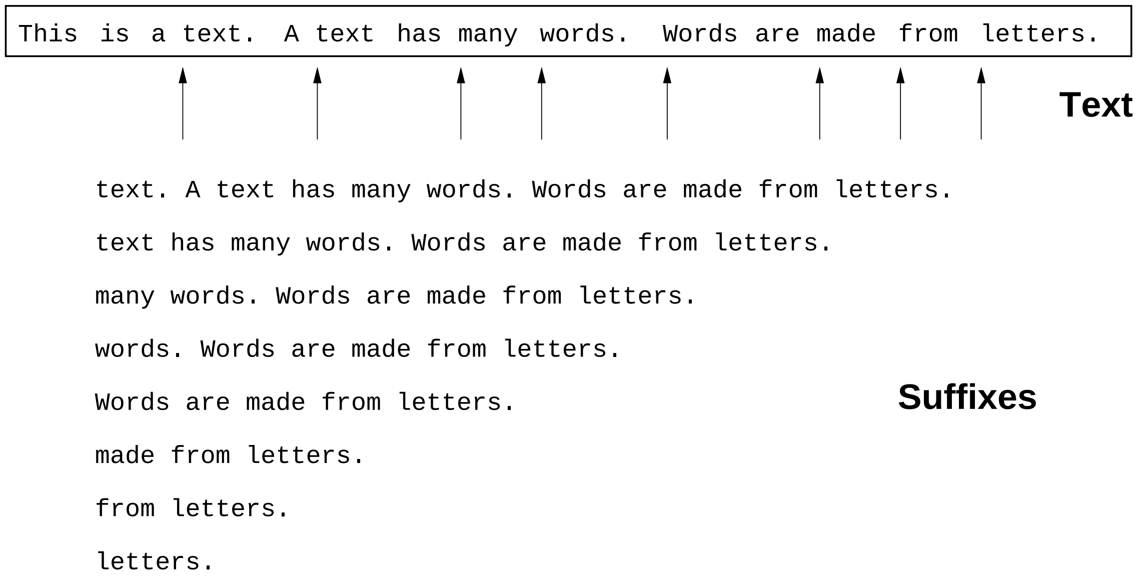
Inverted files assume that the text can be seen as a sequence of words. This restricts somewhat the kinds of queries that can be answered. Other queries such as phrases are expensive to solve. Moreover, the concept of word does not exist in some applications such as genetic databases.

In this section we briefly describe the suffix arrays (also called "PAT arrays") [MM90, GBY91]. Suffix arrays are a space-efficient implementation of suffix trees (see Section 2.7). This type of index allows efficiently answering more complex queries. Its main drawbacks are its costly construction process and that the text must be readily available at query time. This structure can be used to index only words (without stopwords) as the inverted index as well as to index any text character. In general, *index points* are set to the beginnings of the retrievable text positions (e.g. beginnings of words). This makes the suffix array suitable to a wider spectrum of applications, such as genetic databases. However, for word based applications, inverted files perform better unless complex queries are an important issue.

This index sees the text as one long string. Each position in the text is considered as a text *suffix* (i.e. a string that goes from that text position to the end of the text). It is not difficult to see that two suffixes starting at different position are lexicographically different (assume that a character smaller than all the rest is placed at the end of the text). Each suffix is thus uniquely identified by its position. Figure 2.12 exemplifies.

Suffix arrays provide essentially the same functionality as suffix trees at much less space requirements. If the leaves of the suffix tree are traversed in left-to-right order, all the suffixes of the text are retrieved in lexicographical order. A suffix array is simply an array containing all the pointers to the text suffixes listed in lexicographical order, as shown in Figure 2.13. Since they store one pointer per indexed word, the space requirements are almost the same as those for inverted indices (disregarding compression techniques), i.e. close to 40% overhead over the text size. A suffix tree built on a text is 3 to 6 times larger.

While suffix trees are searched as tries, suffix arrays are binary searched. However, almost every algorithm on suffix trees can be adapted to work on suffix arrays at an $O(\log n)$ penalty factor in the



This is a text. A text has many words. Words are made from letters.
Text
text. A text has many words. Words are made from letters.
text has many words. Words are made from letters.
many words. Words are made from letters.
words. Words are made from letters.
Words are made from letters.
made from letters.
from letters.
letters.
Suffixes

Figure 2.12: The sample text with the index points of interest marked. Below, the suffixes corresponding to those index points.

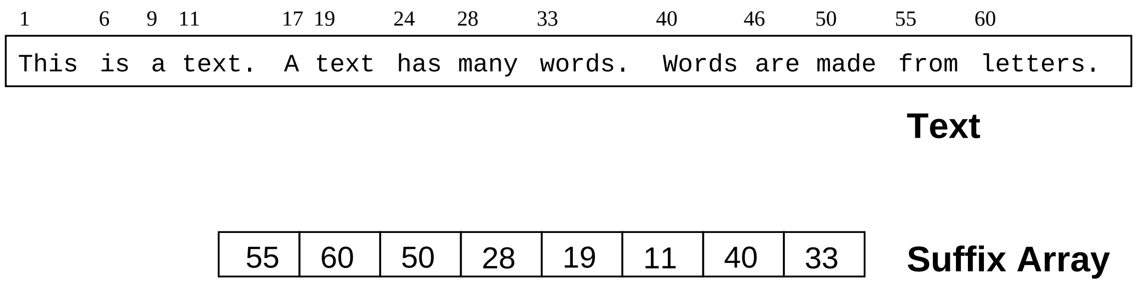


Figure 2.13: The suffix array for the sample text.

time cost. This is because each edge traversal in the suffix tree can be simulated with two binary searches on the suffix array (each subtree of the suffix tree corresponds to an interval in the suffix array). Each time an edge of the suffix trie is followed, we use binary search to find the new limits in the suffix array.

# Chapter 3

# Related Work and Our Contributions

In this chapter we cover all the previous work of interest we could trace on approximate string matching, and place our contribution in its context. We have preferred to present this work as a historical tour, so that we do not only explain the work done but also show how has it been developed. We divide the presentation according to the parts of this thesis.

In the discussion that follows, keep in mind that there may be a long gap between the time where a result is discovered and when it gets finally published in its definitive form. Some apparent inconsistencies can be explained in this form (e.g. algorithms which are "finally" analyzed before they appear). We did our best in the bibliography to trace the earliest version of the works, although the full reference is generally for the final version.

At the beginning of each section we give a taxonomy to help guide the tour. The taxonomy is an acyclic graph where the nodes are the algorithms and the edges mean that the lower work can be seen as an evolution of the upper work (although sometimes the developments are in fact independent).

## 3.1 On-line Searching

As explained in the Introduction, on-line searching is the oldest area in the field of approximate string matching. It means that, by some reason, we do not have any data structure built on the text to speed up the search. Since building such data structures (called "indices") is normally more costly than searching the pattern, it is better to proceed with no index at all. Other (partial) surveys on this matter can be found in [HD80, AG85, GG88, JTU96].

We have covered in Section 2.2 the first and classical solution, which was discovered many times and is based on dynamic programming. The solution to compute the edit distance between two strings is attributable to [NW70, San72, WF74] and others. The first one in converting it into a search algorithm to find a pattern in a text allowing errors was Sellers [Sel80]. As said, this solution is unbeaten in flexibility, but its time requirements are indeed high. A number of improved solutions have been proposed along the years. Some of them work only for the edit distance, while others can still be adapted to other distance functions.

We can divide the work after 1980 in four quite independent areas:

- Algorithms that take advantage of the specific properties of the dynamic programming matrix. This is the oldest area, which inherits directly from the earliest work. Most of the theoretical breakthroughs in the worst case algorithms belong to this category, although only a few of them are really competitive in practice for text searching. The latest practical work in this area dates back to 1992, although there are recent theoretical improvements. The major achievements are $O(kn)$ worst-case algorithms and $O(kn/\sqrt{\sigma})$ average-case algorithms, as well as other (non-practical) improvements on the worst-case.

- Algorithms based on deterministic automata. This area is rather old but it has not been active again until very recently. It is interesting because it gives the best worst-case algorithm ($O(n)$, which is the worst-case lower bound of the problem). However, there is a time and space exponential dependence on $m$ and $k$ that limits its practicality.

- Algorithms that filter the text, quickly discarding text areas which cannot match. This is a new branch of the problem which started after 1990 and continues to be very active. It addresses only the average case. Its major interest is the potential for algorithms that do not inspect all characters. The major theoretical achievement is an algorithm with average cost $O((k + \log_\sigma m)/m\ n)$, which has been proven optimal. In practice, the filter algorithms are the fastest ones too. All of them, however, are limited in their applicability by the error level $\alpha$. Moreover, they need a non-filter algorithm to check the potential matches.

- Algorithms based on exploiting the parallelism of the computer when it works on bits. This is also a new (after 1990) and very active area. The basic idea is to "parallelize" another algorithm using bits. The results are interesting from the practical point of view, and are especially significative when short patterns are involved (typical in text retrieval). They may work effectively for any error level.

In this thesis we have made improvements in all the four areas, especially in the last two (see Figure 3.1). We cover now each area separately. Our comments about the practical performance of the algorithms are based on [JTU96, CL92, Wri94] and in many cases on our own experiments.

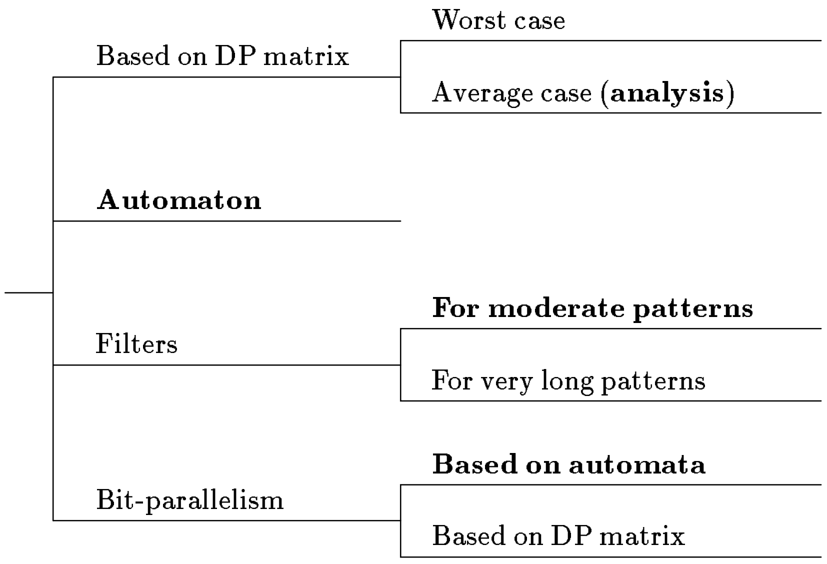


Figure 3.1: Taxonomy of the types of solutions for on-line searching. The areas where we made contributions are in boldface.

### 3.1.1 Taking Advantage of the Dynamic Programming Matrix

[See Figure 3.2]

#### 3.1.1.1 Improving the Worst Case

It is interesting that one of the few worst-case theoretical results on this area is as old as the Sellers [Sel80] algorithm itself. In 1980, Masek and Paterson [MP80] found an algorithm whose worst case is

$$O\left(\frac{mn \log \sigma}{\log^2 n}\right)$$

which is the only improvement known over the $O(mn)$ classical complexity if $k = \Theta(m)$ (although the automaton approach is linear in the text size, it is exponential in $m$ and $k$, while this result is not). The algorithm is based on the Four-Russians technique [ADKF75]. Basically, it first builds a table of solutions of all the possible problems (i.e. portions of the matrix) of size $r$ (for small $r$), and then uses the table to solve the original problem in blocks of size $r$. To achieve the stated result, $\Theta(n)$ extra space is necessary. The algorithm is only of theoretical interest, since as the same authors estimate, it will not beat the classical algorithm for texts below 40 Gb.

In 1983, an article of Ukkonen opened new fruitful areas of development [Ukk85a]. It presented an algorithm able to compute the edit distance between two strings $x$ and $y$ in $O(ed(x,y)^2)$ time, or to check in time $O(k^2)$ whether that distance was $\leq k$ or not. This is the first member of what has been called "diagonal transition algorithms", since it is based in the fact that the diagonals (running from the upper-left to the lower-right cells) of the dynamic programming matrix are monotonically increasing (i.e. $C_{i+1,j+1} \in \{C_{i,j}, C_{i,j}+1\}$). The algorithm is based on computing in constant time the positions where the values along the diagonals are incremented. Only $O(k^2)$ such positions are computed to reach the lower-right decisive cell.

The history of the $O(kn)$ worst-case algorithm is interesting because the results went backwards in many cases. In 1985 and 1986, Landau and Vishkin found the first worst-case time improvements. All of them and the thread that followed were diagonal transition algorithms. In [LV88] they show an algorithm which is $O(k^2 n)$ time and $O(m)$ space, and in [LV89] they obtain $O(kn)$ time and $O(n)$ space.

The main idea of Landau and Vishkin was to adapt to text searching the Ukkonen's diagonal transition algorithm for edit distance [Ukk85a]. Basically, if one thinks on the dynamic programming matrix, it was to be computed diagonal-wise instead of column-wise. They wanted to compute in constant time the next point where the values along a diagonal were to be incremented. This data, called after "matching statistics", are equivalent to knowing which is the longest substring of the pattern that matches the text at some point.

Since a text position was to be reported when row $m$ was reached before incrementing more than $k$ times the values along the diagonal, this gave immediately the $O(kn)$ algorithm. This was done using a suffix tree on the pattern and the text (since they wanted to know the next position where pattern and text were going to differ). This suffix tree was the cause of the huge $O(n)$ extra space, and the $O(1)$ algorithm to find the next diagonal position was slow in practice.

In 1986, Myers found also an algorithm with $O(kn)$ worst-case behavior [Mye86a]. It needed only $O(k^2)$ extra space. However, the reference is a technical report and never went to press (it has been recently included in a larger work [LMS98]).

In 1988, Galil and Giancarlo [GG88] obtained the same time complexity with $O(m)$ space. Basically,

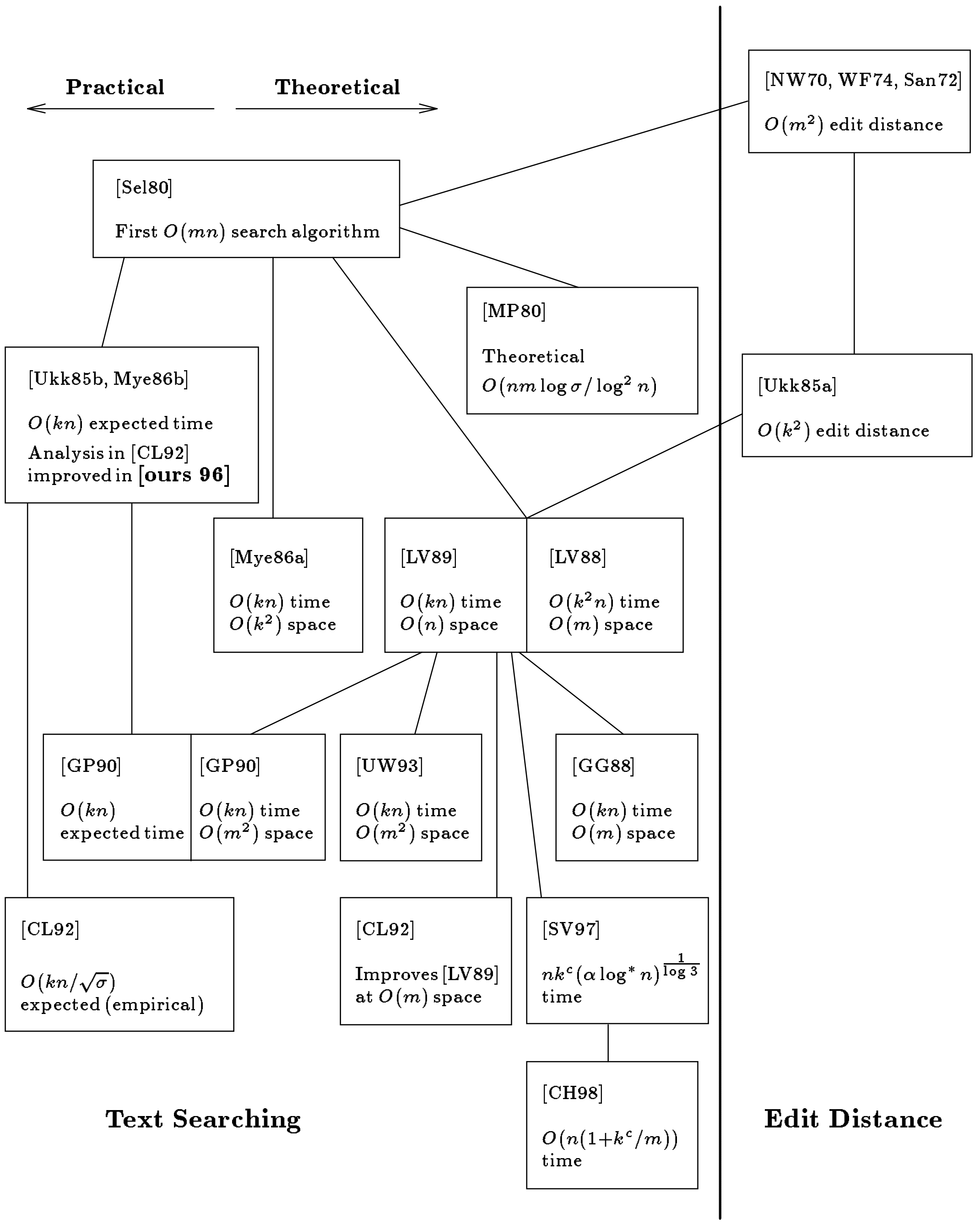


Figure 3.2: Taxonomy of on-line algorithms that take advantage of the dynamic programming matrix.

the suffix tree of the text is built by pieces of size $O(m)$. The algorithm scans the text four times, being even slower than [LV89]. Therefore, the result was of theoretical interest.

One year later, in 1989, Galil and Park [GP90] obtained $O(kn)$ worst-case time and $O(m^2)$ space (worse in theory than in 1988). This time the idea was to build the matching statistics of the pattern against itself (longest match beginning at $i$ versus beginning at $j$, hence the $O(m^2)$ complexity), resembling in some sense the basic ideas of [KMP77]. This algorithm is also slow in practice.

A closely related idea with similar time and space complexity (and similarly slow in practice) was independently discovered by Ukkonen and Wood in 1990 [UW93]. They use a suffix automaton (described in Section 2.8) on the pattern to find the matching statistics, instead of the table.

Finally, in 1992, Chang and Lampe [CL92] improved the algorithm [LV89] to be $O(kn)$ time and $O(m)$ space, this time with better practical performance. This was obtained by using new auxiliary algorithms that appeared in the while. It is said in [CL92] that this modification is the fastest algorithm in practice among those of $O(kn)$ guaranteed performance. However, the algorithm is still not competitive in practice.

In 1997, Sahinalp and Vishkin [SV97] found an $O(nk^c(\alpha \log^* n)^{1/\log 3})$ worst-case time algorithm, which is of theoretical interest if $k = O(m^\epsilon)$ for small $\epsilon$. The constant $c$ is 3 when the pattern has not repeated letters and 8 otherwise. It is based on [LV89], where the set of diagonals to compute is sparsified. This has been recently improved and simplified [CH98] to $O(n(1 + k^c/m))$, where $c = 3$ if the pattern is "mostly aperiodic" and $c = 4$ otherwise (the definition of "mostly aperiodic" is rather technical, and is related to the number of self-repetition that occurs in the pattern). The interest in this development is theoretical.

#### 3.1.1.2 Improving the Average Case

The first improvement to the average case is due to Ukkonen in 1985. The algorithm, a short note at the end of [Ukk85b], improved the dynamic programming algorithm to $O(kn)$ on average. This algorithm has been called later the "cut-off heuristic". The main idea is that, since a pattern does not normally match in the text, the values at each column (looking them from top to bottom) quickly reach $k + 1$ (i.e. mismatch), and that once a cell has a value larger than $k + 1$, the result of the search does not depend on its exact value. A cell is called *active* if its value is at most $k$. The algorithm simply keeps count of which is the last active cell and avoids working on the rest of the cells. Ukkonen conjectured that this algorithm was $O(kn)$ on average, but this was proven only in 1992 by Chang and Lampe [CL92]. As part of this thesis [BYN96b, BYN98d] we find an accurate bound for the probability of matching with errors and use this bound to improve the constant found by Chang and Lampe. It is interesting to notice that, independently of Ukkonen, Myers found almost the same result at about the same time [Mye86b].

An algorithm in [GP90] was based on diagonal transitions but it found the next increment in the diagonal by brute force instead of a guaranteed constant time. This gave an algorithm which was in practice faster than those of guaranteed performance. Myers showed in [Mye86a] that this was $O(kn)$ on average. This algorithm is in fact a variant of [Ukk85b].

The main result of Chang and Lampe [CL92] is a new algorithm based on exploiting a different property of the dynamic programming matrix. They consider again the fact that, along each column, the numbers are normally increasing. They work on "runs" of consecutive increasing cells (a run ends when $C_{i+1} \neq C_i + 1$). They manage to work $O(1)$ per run in the column updating process. Based on empirical observations, they conjecture that the average length of the runs is $O(\sqrt{\sigma})$. Since they use the cut-off heuristic of Ukkonen, their average search time is argued to be $O(kn/\sqrt{\sigma})$. This is, in practice, the fastest algorithm of this class.

There are other algorithms that use properties of the dynamic programming matrix, but they are described in other sections according to their core idea. Hence, the best exponents of this class are

- The $O(mn \log\sigma / \log^2 n)$ algorithm of [MP80], for its theoretically low complexity when $k = \Theta(m)$.
- The $O(n(1 + k^c/m))$ algorithm of [CH98], for its theoretical low complexity when $k = O(m^\epsilon)$.
- The $O(kn)$ time and $O(m)$ space variation of [LV89] as improved by [CL92], for its good theoretical complexity. For small $k$, the algorithm of [Mye86a] could use less space, $O(k^2)$.
- The $O(kn/\sqrt{\sigma})$ expected time algorithm of [CL92], for its good practical behavior.
- The $O(kn)$ average time cut-off heuristic of Ukkonen [Ukk85b], for its simplicity to use as a building block of other algorithms.

In our experimental comparisons we include only the cut-off heuristic of [Ukk85b] and the new algorithm of [CL92]. The others are not of practical interest for typical text searching applications.

### 3.1.2 Searching with a Deterministic Automaton

[See Figure 3.3]

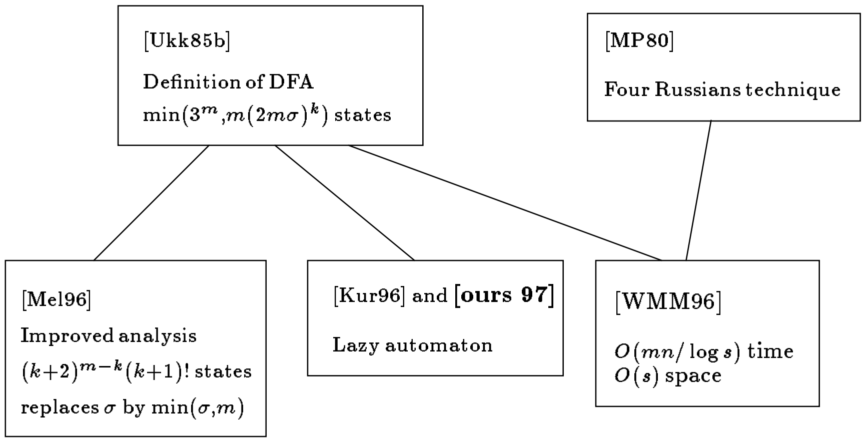


Figure 3.3: Taxonomy of on-line algorithms based on deterministic automata.

Another thread which is also very old but has received less attention is based on reexpressing the problem with the use of an automaton. In Section 2.4 we saw that there is a non-deterministic automaton that solves this problem, so by converting this automaton to deterministic we have immediately the $O(n)$ worst-case time so hardly searched for.

In 1985, Ukkonen proposed the idea of such a deterministic automaton [Ukk85b]. However, an automaton as we have shown in Figure 2.5 was not explicitly considered. Rather, each possible set of values for the columns of the dynamic programming matrix was a state of the automaton. Once the set of all possible columns and the transitions among them were built, the text was scanned with the resulting automaton, performing exactly one transition per character read.

The big problem with this scheme was that the automaton had a potentially huge number of states, which had to be built and stored. To improve space usage, Ukkonen proved that all the elements in the columns that were larger than $k+1$ could be replaced by $k+1$ without affecting the output of the search (the lemma was used in the same paper to design the cut-off heuristic described in Section 3.1.1). This reduced the potential number of different columns. He also showed that adjacent cells in a column differed in at most one. Hence, the column states could be defined as a vector of $m$ incremental values in the set $\{-1, 0, 1\}$.

All this made possible to obtain in [Ukk85b] a nontrivial bound to the number of states of the automaton, namely

$$O(\min(3^m, m(2m\sigma)^k))$$

which, although much better than the obvious $O((k+1)^m)$, was still very bad except for short patterns or very low error levels. The resulting space complexity of the algorithm was $m$ times the above value. This exponential space complexity was to be added to the $O(n)$ time complexity, as the preprocessing time to build the automaton.

As a final comment, Ukkonen suggested that the columns could be only partially computed up to, say, $3k/2$ entries. Since he conjectured (and later was proved in [CL92]) that the columns of interest were $O(k)$ on average, this would normally not affect the algorithm, though it will reduce the number of possible states. If at some point the states not computed were really needed, the algorithm would compute them by dynamic programming.

It was not until 1992 that Wu, Manber and Myers looked again into this problem [WMM96]. The idea was to trade some time for space using a Four Russians technique [ADKF75]. Since the columns could be expressed using only values in $\{-1, 0, 1\}$, the columns were partitioned into blocks of $r$ cells (called "regions") which took $2r$ bits each. Instead of precomputing the transitions from a whole column to the next, the transitions from a region to the next region in the column were precomputed. Since the regions were smaller than the columns, much less space was needed. The total amount of work was $O(m/r)$ per column in the worst case, and $O(k/r)$ on average. The space requirement was exponential in $r$. By using $O(n)$ extra space, the algorithm was $O(kn/\log n)$ on average and $O(mn/\log n)$ in the worst case. Notice that this is in the same trend of [MP80], although it is much more practical.

In 1996, Kurtz [Kur96] proposed another way to reduce the space requirements to at most $O(mn)$. It is an adaptation of [BYG94], who first proposed it for the Hamming distance (as we cover later). The idea was to build the automaton in lazy form, i.e. build only the states and transitions actually reached in the processing of the text. The automaton starts as just one initial state and the states and transitions are built as needed. By doing this, all those transitions that Ukkonen considered that were not necessary were not built in fact, without need to guess. The price was the extra overhead of a lazy construction versus a direct construction, but the idea pays off. Kurtz also proposed to have built only the initial part of the automaton (which should be the most commonly traversed states) to save space.

In 1995, Melichar [Mel96] considered again the finite automaton, this time as the deterministic version of our non-deterministic automaton of Figure 2.5 (he also described automata for the Hamming distance, for transpositions, and others). By using properties of this automaton, he improved the bound of [Ukk85b] to

$$O(\min(3^m, m(2mt)^k, (k+2)^{m-k}(k+1)!)$$

where $t = \min(m+1, \sigma)$. The space complexity and preprocessing time of the automaton is $t$ times the above formula. Melichar also conjectured that this automaton is bigger when there are periodicities in the pattern. It is rather common in text searching that periodic patterns are more problematic, and fortunately periodic patterns are uncommon in typical text searching.

As part of this thesis [Nav97b], we improved the implementation of the lazy automaton and studied its growth as a function of $m$, $k$ and $n$ (this last value makes sense for the lazy automaton only). We studied replacement policies to be able to work with bounded memory. We also tested this idea against the best known algorithms and found it competitive for short to medium patterns and medium error levels (but it is not the fastest one nowadays, although in some parameter ranges it was the fastest when we tried it).

Nowadays, the best exponents of this trend are [WMM96] (which is a very practical and competitive algorithm), and [Kur96, Nav97b] (since the optimized implementations are competitive when the number of states is not too large). We test both algorithms in this thesis.

### 3.1.3 Filtering

[See Figure 3.4]

This is a much newer trend, currently very active. It is based on finding fast algorithms to discard large areas of the text that cannot match, and apply another algorithm in the rest. As explained in Section 2.5, a new factor plays its role here: the maximum error level up to where the filters leave out enough text to be worthwhile.

Although some of the best existing algorithms belong to this class, it is important to recall that these algorithms need another one to verify the matching positions. Therefore, it is still interesting to pursue on other type of algorithms, since the best combination is possibly a marriage between a filtering algorithm and an algorithm capable of verification.

We divide this area in two parts: moderate and very long patterns. Typical text searching deals with the first part. The algorithms for the two areas are normally different, since more complex filters are worthwhile only on longer patterns.

#### 3.1.3.1 Moderate Patterns

This area started in 1990, where Tarhio and Ukkonen [TU93] published an algorithm that tried to use Boyer-Moore-Horspool techniques [BM77, Hor80] to filter the text. The idea was that a diagonal of the dynamic programming matrix must have $k$ or less mismatches (increments) in order to match. Notice that, inside a single diagonal, we can think only in terms of mismatches, since insertions and deletions affect other diagonals. Thanks to insertions/deletions a diagonal may not increase its value even in case of a mismatch, but it cannot reach $k+1$ if there are not $k$ or more mismatches. Once it was determined that a given diagonal (i.e. text position) matched or not, they moved to the next diagonal. However, some diagonals could be skipped if they did not match in enough places with the current diagonal (recall that neighbor diagonals share characters). This was solved in constant time by some preprocessing of the pattern. The algorithm is competitive in practice for text searching.

In 1991, Jokinen, Tarhio and Ukkonen [JTU96] adapted a previous filter for the $k$-mismatches problem [GL89]. The filter is based on the simple fact that inside any match with at most $k$ errors there must be at least $m-k$ letters belonging to the pattern. The filter did not care about the order of those letters. This is a simple version of [CL94] (see next section), with less filtering efficiency but simpler implementation. They slid a window of variable size over the text. As part of this thesis

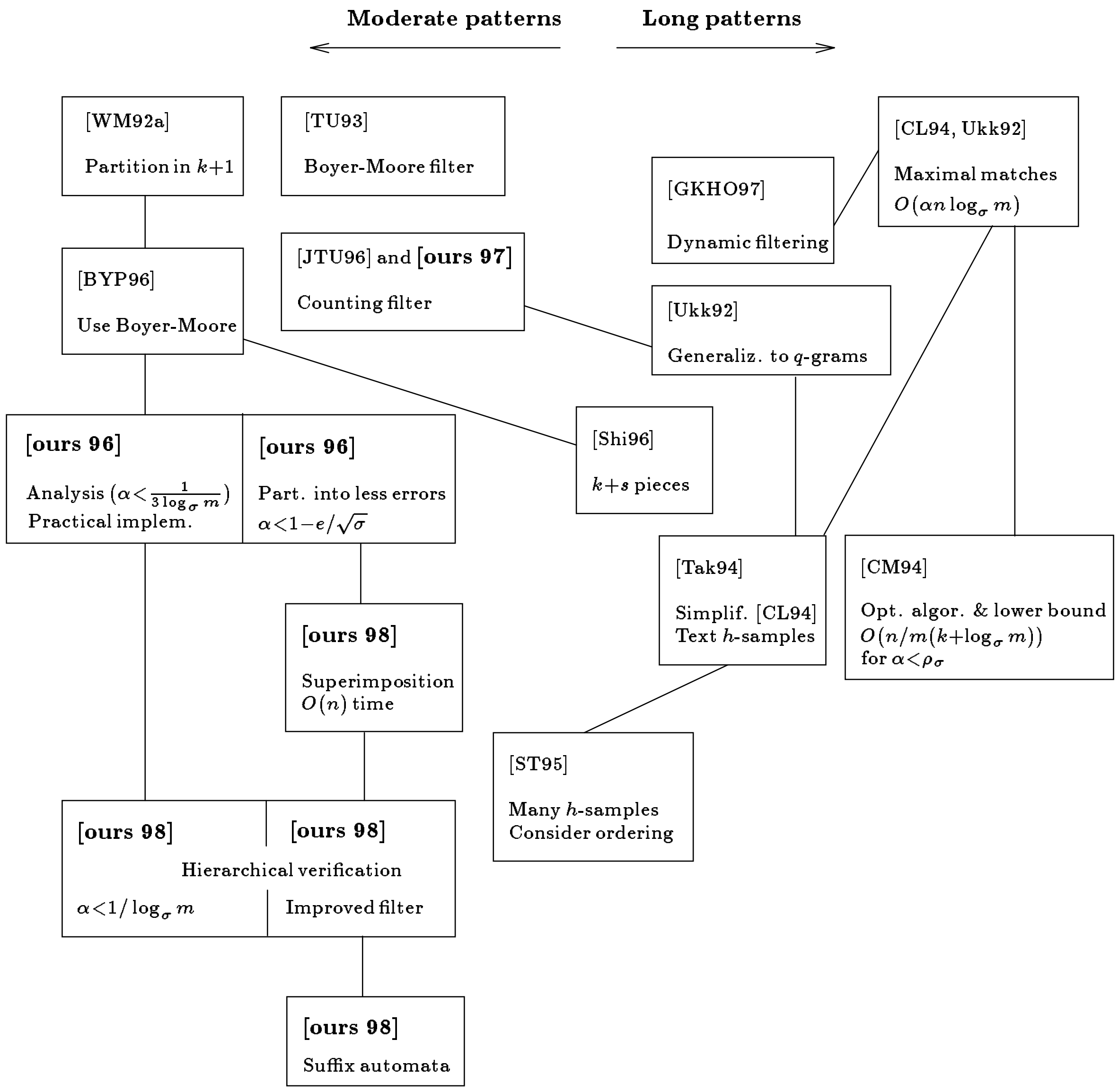

Moderate patterns
Long patterns
[WM92a] Partition in $k+1$
[TU93] Boyer-Moore filter
[CL94, Ukk92] Maximal matches $O(\alpha n \log_\sigma m)$
[GKHO97] Dynamic filtering
[JTU96] and [ours 97] Counting filter
[BYP96] Use Boyer-Moore
[Ukk92] Generaliz. to $q$-grams
[ours 96] Analysis ($\alpha < \frac{1}{3\log_\sigma m}$) Practical implem.
[ours 96] Part. into less errors $\alpha < 1-e/\sqrt{\sigma}$
[Shi96] $k+s$ pieces
[Tak94] Simplif. [CL94] Text $h$-samples
[CM94] Opt. algor. & lower bound $O(n/m(k+\log_\sigma m))$ for $\alpha < \rho_\sigma$
[ours 98] Superimposition $O(n)$ time
[ST95] Many $h$-samples Consider ordering
[ours 98] [ours 98] Hierarchical verification $\alpha < 1/\log_\sigma m$ Improved filter
[ours 98] Suffix automata

Figure 3.4: Taxonomy of on-line filtering algorithms.

[Nav97a] we rediscovered the algorithm (observe that it did not appear in press until 1996), this time with a simplified version of fixed-size window and with analytical bounds on the maximum error level where it could be applicable (we also adapted the algorithm to multipattern search but that is explained later). The resulting algorithm is competitive in practice for text searching.

In 1992, a very simple filter was proposed by Wu and Manber [WM92a] (among many other ideas of that work). The basic idea is in fact very old [Riv76]: if a pattern is cut in $k+1$ pieces, then at least *one* of the pieces must appear unchanged in an approximate occurrence. This is evident, since $k$ errors cannot alter the $k+1$ pieces. The proposal was then to split the pattern, search the pieces in parallel, and check the neighborhood of the exact matches of the pieces. Since they were embedded in a framework of bit-parallel algorithms (see Section 2.6), they adapted one such algorithm for this task, so that they searched the text at $O(mn/w)$ time complexity and verified the matches of the pieces. In the same year, Baeza-Yates and Perleberg [BYP96] suggested that better algorithms could be used for the multipattern search: at least an Aho-Corasick machine [AC75] to guarantee $O(n)$ search time (excluding verifications), or even a Boyer-Moore algorithm adapted to multipattern search, such as [CW79]. However, no more attention was paid to this idea until this thesis.

In 1996, we resumed the work on this filter. First, we adapted to multipattern exact search a simple variation of the Boyer-Moore-Sunday algorithm [Sun90] (an algorithm for exact string matching). The adaptation uses a trie for the patterns and computes a pessimistic shift table among all the patterns, and works well for a moderate number of them. Then, we implemented the filter using the multipattern Sunday algorithm for the search task. We found that this algorithm was the fastest known one when the error level was moderate (say, $\alpha < 1/4$ on English text). These results were published in [BYN96b, BYN96a] and the 1996 version of [BYP96] (which also contains and independent analysis of the algorithm). We also analyzed this scheme and found that the filter worked at least for $\alpha < 1/(3\log_\sigma m)$ (which is similar to other limits in this area). Before that limit, the algorithm can run in $O(n)$ guaranteed search time, and our Sunday implementation can run at $O(\alpha n)$ time in the best case (which is optimal, see [CM94] in the next section). There is no closed expression for the average case [BYR90].

In Section 3.1.4 we describe a new bit-parallel algorithm developed as part of this thesis, which is based on the NFA described in Section 2.4. As the algorithm is limited to short patterns, many techniques are developed to cope with longer patterns [BYN96b, BYN96a, BYN98d]. One of them is a filter based in partitioning the pattern in subpatterns. It is shown that if a pattern is partitioned in $j$ pieces, the pieces can be searched with $\lfloor k/j \rfloor$ errors. This is a generalization of the idea of partitioning the pattern in $k+1$ pieces, and has been mentioned in an earlier work of Myers [Mye94] in a totally different context (indexed searching, which is covered later). In this thesis we give the definitive and most general form to this "partitioning lemma". The idea is thus to partition the pattern and the number of errors in as few pieces as possible so that the subpatterns can be searched with the bit-parallel algorithm. We show that some of the pieces can be in fact "superimposed" [BYN98d] to reduce the number of pieces to search (the superimposition is based on allowing many letters to match the horizontal edges of the automaton). We also develop a hierarchical verification technique that abandons false alarms as quickly as possible [NBY98b]. This allows using our filter for $\alpha < 1 - e/\sqrt{\sigma}$ (in practice the $e$ can be replaced by 1). It is important to notice that this filter is *not* sublinear, but $O(n)$ search time in the best case (it really cannot skip characters). However, $O(n)$ average search time is also very appealing, especially for not so low error levels.

Apart from the theoretical filter of [CM94] (see next section on filters for long patterns), this is the first filter that reduces the problem to *less* errors instead of to *zero* errors. An interesting observation is that it seems that all filters that partition the problem into exact search can be applied for $\alpha = O(1/\log_\sigma m)$, and that in order to improve that bound we must partition the problem into (smaller) approximate searching subproblems. On the other hand, we present strong evidence in this thesis to show that the limit $\alpha = 1 - 1/\sqrt{\sigma}$ is an absolute barrier for every filtration algorithm, since there are simply too many *real* matches, and even the best filter must check real matches.

We also apply the hierarchical verification technique to the simple scheme that partitions the pattern in $k+1$ pieces [NBY98d]. With this technique, the applicability of the filter is increased, being even $\alpha < 1/2$ for $\sigma = 32$ (an asymptotical bound of $\alpha < 1/\log_\sigma m$ is proved). This covers almost all the cases of interest in text searching. It is important to observe that our filter degrades (slowly) as the pattern length grows, unlike other filters. This makes it especially suitable for typical text searching, where the more complex filters are not competitive.

In 1998 we adapted the work of [NR98a] (which is not part of this thesis and is briefly described in Section 2.8). In this algorithm an automaton that recognizes a pattern (with no errors) is transformed so that it recognizes any suffix of the pattern and is simulated with bit-parallelism. The modification is to add a new state with $\varepsilon$-transitions to all the other states. This automaton is used as part of an algorithm for exact string matching. In [NR98b] we take our bit-parallel automaton of [BYN96b] and modify it to recognize any suffix of the pattern allowing errors. A slight modification of the exact matching algorithm serves as a filter to search allowing errors without inspecting all the text characters. The result is competitive for very low error levels, and it is the fastest in some cases, such as for DNA on intermediate pattern lengths and a few errors.

#### 3.1.3.2 Very Long Patterns

In 1990, Chang and Lawler [CL94] presented two algorithms. A first one is $O(kn)$ in the worst case (it uses [LV89] as its verifying algorithm), but $O(n)$ on average provided $\alpha < 1/(\log_\sigma m + O(1))$. The constants are involved, but practical figures are $\alpha \leq 0.35$ for $\sigma = 64$ or $\alpha \leq 0.15$ for $\sigma = 4$, The idea is to build a suffix tree on the pattern and use it to derive the matching statistics in constant time (i.e. longest pattern substring matching $T_{i\ldots}$). Given a text position, they repeat $k$ times the process of concatenating the longest possible pattern substrings matching the text. If after performing those $k$ errors they did not cover $m-k$ text characters, a match is not possible. Notice that they do not force that the substrings of the pattern matched were ordered in the pattern.

A sublinear[1] expected time algorithm was possible when $k = o(m/\log m)$, by splitting the text in fixed blocks of size $(m-k)/2$ and starting the verifications only at block boundaries. This is because any match had to contain some block completely. Despite their good complexity, the algorithms are useful only for long patterns, which are rare in text searching.

In 1992, Ukkonen [Ukk92] independently rediscovered some of the ideas of Chang and Lampe. He presented two filtering algorithms, one of which (based on what he called "maximal matches") is similar to the linear expected time algorithm of [CL94]. However, in the same paper he makes the first reference to "$q$-grams" for on-line searching (we will see older references in indexed searching). A $q$-gram is a substring of length $q$ (see also Section 2.10). A filter was proposed based on counting the number of $q$-grams shared between the pattern and a text window (considering repetitions). A pattern of length $m$ has $(m-q+1)$ $q$-grams (they overlap). The number of repetitions of each $q$-gram had to be close in the pattern and the text window for a match to be possible. Notice that this is a generalization of the simple counting filter of [JTU96] (which corresponds to $q = 1$).

[1]Refer to Section 2.5 for the concept of sublinearity.

In 1994, Takaoka [Tak94] presented a simplification of [CL94]. He considered $h$-samples of the text (which are non-overlapping $q$-grams of the text taken each $h$ characters, for $h \geq q$. The idea was that if one $h$-sample was found in the pattern, then a neighborhood of the area was verified. This is true for $h = \lfloor (m-k-q+1)/(k+1) \rfloor$. The average complexity of the search is $O(kn \log_\sigma(m)/m)$, which is sublinear for $k$ small enough.

Sutinen and Tarhio generalized the Takaoka filter in 1995 [ST95], improving its filtering efficiency. This is the first filter that takes into account the relative positions of the pattern pieces that match in the text (all the previous matched pieces of the pattern in any order). The generalization is to force that $s$ $q$-grams of the pattern match (not just one). The pieces must conserve their relative ordering in the pattern and must not be more than $k$ characters away from their correct position. In this case, the sampling step is reduced to $h = \lfloor (m-k-q+1)/(k+s) \rfloor$. The pattern is pruned at $(k+2)h$ characters and $k+2$ sets of $h+2k$ $q$-grams are extracted (by symmetrically overlapping with previous and next areas). Those sets are called $Q_i$. All the contiguous sequences of $k+2$ text $h$-samples are considered. If at least $s$ of the $k+2$ text samples are found in their respective $Q_i$ set of pattern $q$-grams, the area is verified. This reduces the problem to counting mismatches, and Sutinen and Tarhio use the algorithm [BYG92] for this matter. The resulting algorithm is $O(k^2 n \log_\sigma(m)/(mw))$ on average, for $q = O(\log_\sigma n)$. The algorithm works well for long patterns, although with $s = 2$ can be reasonably applied to typical text searching.

It looks like $O(kn \log_\sigma(m)/m)$ is the best complexity achievable by using filters, and that it will work only for $k = O(m/\log m)$, but in 1994 Chang and Marr obtained at the same time an algorithm which was

$$O\left(\frac{k + \log_\sigma m}{m} \; n\right)$$

for $k < \rho_\sigma m$, where $\rho_\sigma$ depends only on $\sigma$ and for very large $\sigma$ it tends to $1 - e/\sqrt{\sigma}$. At the same time, they proved that this was a lower bound for the average complexity of the problem (and therefore their algorithm was optimal on average). This is a major theoretical breakthrough.

The lower bound is obtained by taking the maximum (or sum) of two simple facts: the first one is the $O(n \log_\sigma(m)/m)$ bound of [Yao79] for exact string matching, and the second one is the obvious fact that in order to discard a block of $m$ text characters, at least $k$ should be examined to find the $k$ errors (and hence $O(kn/m)$ is a lower bound). It is more impressive that an algorithm with such complexity was found.

The algorithm is a variation of the sublinear one of [CL94]. It is of polynomial space in $m$, i.e. $O(m^t)$ space for some constant $t$ which depends on $\sigma$. It is based on splitting the text in many substrings of size $\ell = t \log_\sigma m$. Instead of searching the longest exact matches of the pattern in the beginning of blocks of size $(m-k)/2$, it searches the text substrings of length $\ell$ in the pattern, allowing errors. The longest matches allowing errors inside $P$ are precomputed for every $\ell$-tuple (hence the $O(m^t)$ space). Though very interesting theoretically, the algorithm is not practical except for very long patterns.

It is interesting to notice that the limit $k \leq m(1 - e/\sqrt{\sigma})$ appears in our work too, as a firm limit of any filtering mechanism. Chang and Lawler proved an asymptotic result, while in this thesis we prove a better bound. As they used this analysis to prove that the cut-off heuristic of [Ukk85b] was $O(kn)$ on average, this explains that we could find a better constant redoing the same analysis and using our own bounds.

In 1996 Shi [Shi96] proposed to extend the idea of the $k+1$ pieces (explained in the previous section on short patterns) to $k+s$ pieces, so that at least $s$ pieces must match. He compared his filter against the simple one, finding that the filtering efficiency was improved. However, this improvement will be noticeable only for long patterns. Moreover, the on-line searching efficiency is degraded because the pieces are shorter (which affects any Boyer-Moore-like search), and because the verification logic is more complex. However, the scheme can be more interesting for indexed searching (which was the context in which Shi presented the paper), and we review it in that context later.

Also in 1996, a general method to improve filters was developed [GKHO97]. The idea is to mix the phases of filtering and checking, so that the verification of a text area is abandoned as soon as the combined information from the filter (number of guaranteed differences left) and the verification in progress (number of actual differences seen) shows that a match is not possible. As they show, however, the improvement occurs in a very narrow area of $\alpha$. This is a general phenomenon of the statistics of this problem that we are going to analyze in detail in this thesis. They also tune the analysis of [CL94].

The best filter for typical text searching is nowadays our optimized version of the $k+1$ pieces filter [NBY98d]. Other competitive filters are [TU93, ST95, JTU96]. All these filters are included in our empirical comparisons. On the other hand, we are not covering in this thesis the area of long patterns (typical in computational biology), where most of the mentioned filters do not work well, and other are better [CM94, CL94, Ukk92, ST95]. A major theoretical achievement of this area is the filter [CM94], which is proven to be optimal on the average.

### 3.1.4 Bit-Parallel Algorithms

[See Figure 3.5]

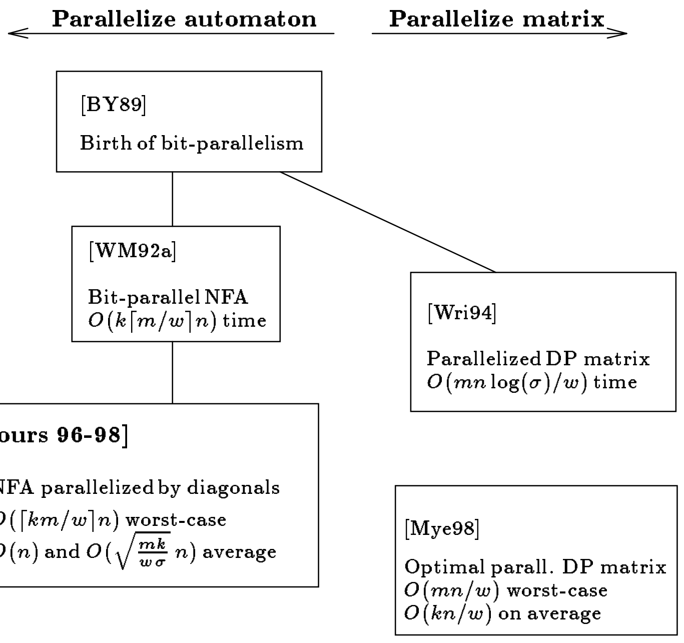

Parallelize automaton
Parallelize matrix
[BY89]
Birth of bit-parallelism
[WM92a]
Bit-parallel NFA
$O(k\lceil m/w\rceil n)$ time
[Wri94]
Parallelized DP matrix
$O(mn\log(\sigma)/w)$ time
[ours 96-98]
NFA parallelized by diagonals
$O(\lceil km/w\rceil n)$ worst-case
$O(n)$ and $O(\sqrt{\frac{mk}{w\sigma}}\,n)$ average
[Mye98]
Optimal parall. DP matrix
$O(mn/w)$ worst-case
$O(kn/w)$ on average

Figure 3.5: Taxonomy of on-line bit-parallel algorithms.

Bit-parallelism [BY91] was born in the PhD. Thesis of Baeza-Yates [BY89]. As explained in Section 2.6, the idea is to simulate parallelism using the bits of the computer word (whose number of bits we denote by $w$). A simple algorithm is chosen, its work is parallelized, and its complexity is reduced (ideally by a factor of $w$). Hence, in this section we find elements which strictly could belong to other sections, since we parallelize other algorithms. There are two main trends: parallelize the work of the non-deterministic automaton that solves the problem (Figure 2.5), and parallelize the work of the dynamic programming matrix.

#### 3.1.4.1 Parallelizing Non-deterministic Automata

The first bit-parallel algorithm was the exact string matching algorithm Shift-Or, which parallelizes a non-deterministic automaton. Despite that the algorithm is not competitive against the fastest ones, it has been the basis for a lot of development that followed it, especially because of its ability to handle some extended patterns. The Shift-Or algorithm was published in [BY89, BYG92], where it was extended to handle classes of characters, multiple patterns, and mismatches.

In 1992, Wu and Manber [WM92a] published a number of ideas that had a great impact in the future of text searching. They first extended the Shift-Or algorithm to handle *wild cards* (i.e. allow an arbitrary number of characters between two given positions in the pattern), and regular expressions (which in fact can be considered the most flexible extended pattern). What is of more interest to us is that they presented a simple scheme to combine any of the preceding extensions to approximate string matching. The idea is to simulate using bit-parallelism the NFA of Figure 2.5, so that each row of the automaton fits in a computer word (each state is represented by a bit). The row $i$ represents having matched the pattern with $i$ errors. For each new text character, all the transitions of the automaton are simulated using bit operations among the $k+1$ computer words. Notice that all the $k+1$ computer words have the same structure (i.e. the same bit is aligned to the same text position). The cost of this simulation is $O(k\lceil m/w\rceil n)$ in the worst and average case, which is $O(kn)$ for patterns typical in text searching (i.e. $m \leq w$). This is a perfect speedup over the serial simulation of the automaton, which would cost $O(mkn)$ time. Notice that for short patterns, this is competitive to the best worst-case algorithms.

Thanks to the simplicity of the construction, the rows of the pattern can be changed by a different automaton. As long as they are able to solve a problem for exact string matching, they make $k+1$ copies of the resulting computer word, perform the same operations in the $k+1$ words (plus the arrows that connect the words) and they have an algorithm to find the same pattern allowing errors. Hence, they are able to perform approximate string matching with sets of characters, wild cards, and regular expressions. They also allow some extensions typical of approximate searching: a part of the pattern can be searched with errors and another may be forced to match exactly, and different costs of the edit operations can be accommodated (including not allowing some of them). Finally, they are able to search a set of patterns at the same time, but this capability is very limited (since all the patterns must fit in a computer word).

The great flexibility obtained encouraged the authors to build a software called *Agrep* [WM92b], where all these capabilities are implemented (although some particular cases are solved in a different manner). This software has been taken as a reference in all the subsequent research.

In 1996, as part of this thesis, we presented a new bit-parallel algorithm able to parallelize the computation of the automaton even more [BYN96b, BYN96a, BYN98d]. The classical dynamic programming algorithm can be thought of a column-wise "parallelization" of the automaton [BY96], and Wu and Manber [WM92a] proposed a row-wise parallelization. Neither algorithm was able to increase the parallelism because of the $\varepsilon$-transitions of the automaton, which caused what we call *zero-time* dependencies. That is, the current values of two rows or two columns depend on each other, and hence cannot be computed in parallel. We were able to find the bit-parallel formula for a *diagonal* parallelization. That is, we packed the states of the automaton along diagonals instead of rows or columns (notice that this is totally different from the diagonals of the dynamic programming matrix). This idea was mentioned in [BY91] but no bit-parallel formula was found. The resulting algorithm is $O(\lceil km/w \rceil n)$ worst case time, whose improvement over [WM92a] is especially noticeable for short patterns (where our algorithm is $O(n)$). This algorithm is the *fastest* known algorithm for short patterns, except for low error levels where our filtration algorithm [NBY98d] is faster. Moreover, many of the extended patterns shown in [WM92a] can also be searched in this scheme.

We proposed many techniques to handle longer patterns. The simplest one is to partition the automaton in many computer words. For moderate error levels, not all the words have to be updated for each text character. This is thanks to the Ukkonen cut-off technique, which shows that the first (left to right) $O(k)$ diagonals of the automaton have active states, on average. Other partitioning and automata superimposition techniques have been already explained as filtering approaches. In 1998 [NBY98b] we developed improved techniques of register usage and improved verifiers for the filters, as well as an optimal algorithm to combine all the possible partitioning techniques. A complete theoretical analysis and experiments were performed, showing that on average the resulting algorithm was $O(n)$ for low error levels, $O(\sqrt{mk/(w\sigma)}\ n)$ for moderate error levels, and $O(k(m-k)n/w)$ otherwise. The result is that the algorithm is not only the fastest one for very short patterns, but also for patterns of any length provided the error level keeps moderate. The only faster algorithm is also part of this thesis (the $(k+1)$-pieces filter of [NBY98d]), and improves this one for moderate error levels and not very long patterns. However, other filters could improve our algorithm for very long patterns, although this is outside the scope of typical text searching.

#### 3.1.4.2 Parallelizing the Dynamic Programming Matrix

In 1994, Wright [Wri94] presented a first work using bit-parallelism on the dynamic programming matrix. The idea was to consider *secondary diagonals* (i.e. those that run from the upper-right to the bottom-left) of the matrix. He used some properties of the matrix to find that the cells could be represented with two bits (for the values $\{-1, 0, 1\}$), and managed to represent the process using three contiguous diagonals and to partially parallelize the computation. The parallelization works by performing in parallel a number of comparisons of the pattern versus the text, and then using the vector of the results of the comparisons to update many cells of the diagonal in parallel. Since he has to store characters of the alphabet in the bits, his algorithm is $O(nm\log(\sigma)/w)$ in the worst and average case. This was competitive for very small alphabets (e.g. DNA).

In 1998 (after our algorithm [BYN96b] was published), Myers [Mye98] found a better way to parallelize the computation of the dynamic programming matrix. Myers was also able to represent the differences along columns instead of the columns themselves, so that two bits per cell were enough (in fact this algorithm can be seen as the bit-parallel implementation of the automaton which is made deterministic in [WMM96], see Section 3.1.2). The parallelization has optimal speedup, and the time complexity is $O(kn/w)$ on average and $O(mn/w)$ in the worst case. A problem which was very similar to that of breaking the $\varepsilon$-dependences was solved (and the solution formula has an interesting resemblance to ours, despite the models being very different). The result is an algorithm that uses better the bits of the computer word. The formula is a little more complex than that of [BYN96b] and hence the algorithm is a little slower, but it is can accommodate longer patterns using less computer words, improving all the other algorithms (including ours) for high error levels and not very long patterns. As it is difficult to improve over $O(kn)$ algorithms, this algorithm may be the last word with respect to asymptotic efficiency of parallelization. As it is now common to expect in bit-parallel algorithms, this scheme is able to search some extended patterns as well.

At the present time, most of the bit-parallel algorithms are worthwhile in practice. [WM92a] is unbeaten in flexibility (although it is the less efficient scheme nowadays). [Wri94] is efficient in small alphabets (we know no other algorithm that improves as $\sigma$ decreases), but it is not the fastest nowadays (it is slower than [Mye98] even on a binary alphabet). Our algorithms and [Mye98] are the fastest in different areas. We consider all these algorithms in our experiments.

## 3.2 Variants on the On-line Problem

As explained in the Introduction, there are a lot of variants of the basic problem. We can classify them in three groups: variants on the kind of pattern to search, variants on the text form, and variants of the distance function. We explain briefly these alternatives (which can, in principle, be combined).

**Varying the type of pattern:** instead of being just a sequence of characters, the pattern could be a more general expression, for instance a regular expression. These are called "extended patterns". Another interesting variant is when the pattern is in fact a set of strings and we want to find all their occurrences.

**Varying the type of text:** a second alternative is to consider that the text could be generalized. A first generalization is to consider classes of characters in the text (this has applications, for instance, to computational biology). Computational biology is also interested in approximate structure matching (e.g. finding molecules whose spatial distribution is close to a query). Other interesting extensions is to consider a multidimensional text (and pattern), which has applications to image processing and computer vision; a non-linear text (hypertext); a compressed text; and many others.

**Varying the distance function:** as we have selected the Levenshtein or edit distance as our main focus (since we are mainly interested in text searching), all the other distances such as Hamming, longest common subsequence, episodes, reversals and others used in computational biology, etc. are considered "variants". As explained in the Introduction, the problem becomes very different when the distance function is modified, and therefore it is difficult to develop general algorithms. We introduced some of the most interesting (to us) distance functions in Section 2.1, but the spectrum is very broad and is out of the scope of this work.

We do not address in general any variant of the classical problem in this thesis. However, some exceptions will be made. First, many of our algorithms are able to handle extended patterns and some restricted alternatives of distance functions (related to assigning different costs to the edit operations), and we mention throughout the thesis when this is the case. Second, we will devote one chapter to on-line multipattern searching, since we believe that, together with extended patterns, this is one of the most important and practical variants on the classical problem.

We present now a historical background on the variants which are considered in this thesis.

### 3.2.1 Extended Patterns and Different Costs

A possible extension of the search pattern is what is called "classes of characters" [BY89] (as well as "limited expressions" in [WMM96]), see Section 2.6. The idea is that the pattern does not belong to $\Sigma^*$ but it belongs to $\mathcal{P}(\Sigma)^*$. That is, each pattern position is a *set* of characters. Such pattern $P = p_{1..m}$ matches $T_{i+1..i+m}$ if $T_{i+j} \in p_j$ for all $j \in 1..m$. In a more colloquial style, a number of interesting cases follow:

- range of characters (e.g. `t[a-z]xt`, where `[a-z]` means any letter between `a` and `z`);
- arbitrary sets of characters (e.g. `t[aei]xt` meaning the words `taxt`, `text` and `tixt`);
- complements (e.g. `t[∼ab]xt`, where `∼ab` means any single character except `a` or `b`; `t[∼a-d]xt`, where `∼a-d` means any single character except `a`, `b`, `c` or `d`);
- arbitrary characters (e.g. `t·xt` means any character as the second character of the word);
- case insensitive patterns (e.g. `Text` and `text` are considered as the same words).

There is some theoretical work on searching allowing errors and classes of characters and other extensions, e.g. [FP74, Abr87]. However, those works are of theoretical interest mainly.

The real practical breakthrough occurred with the birth of bit-parallelism [BYG92]. A good point of almost all bit-parallel algorithms is that they can handle classes of characters at virtually zero extra cost. In Section 2.6 we explain how to do it for exact searching in the Shift-Or algorithm, just by changing the preprocessing step and using the same algorithm. The adaptation is exactly the same in all cases.

In their 1992 paper [WM92a], Wu and Manber showed that the bit-parallel approach could be extended to allow errors and *at the same time* to support a number of extended patterns (as we have seen in Section 3.1.4). Those extensions included not only classes of characters, but also arbitrary number of wild cards ($x\#y$ matches $xzy$ for any string $z$) and combining parts of the pattern that are to be matched exactly and others that are allowed to have errors. As we will see in this thesis, our faster versions of this algorithm support most of these options.

The most general pattern supported by the approach of [WM92a] is the regular expression, which can be searched allowing errors. There are other previous works able to solve the same problem, such as [WS78, MM89, WMM95], but according to [WM92a] they are much slower on typical text retrieval queries.

Finally, [WM92a] also shows how to assign different costs to the different edit operations (insertion, deletion, substitution), as well as how to forbid some operation.

There is no specific chapter for these issues in the thesis. Rather, when we present our on-line algorithms we point out how can they handle more general patterns or different costs in the distance function.

Some of the theoretical works mentioned here can handle classes of characters in the text as well, and in general all bit-parallel algorithms can manage to allow classes of characters in the text. If a text special character $C$ is defined as a subset of the alphabet, then the table of the special character is the bitwise-or of the tables of all the characters in $C$ (the "or" here assumes that 1 means matching, otherwise it is "and"). Hence this is a text variant generally supported with bit-parallelism.

### 3.2.2 Multiple Patterns

[See Figure 3.6]

Another different kind of "extension" to the pattern is to consider multipattern matching. A set of patterns is given and we are required to find all their occurrences in the text. This problem has many applications, for instance

- Spelling: instead of searching incorrect words in the dictionary to find the most likely variants, we may search the dictionary in the text, hopefully at much less cost.

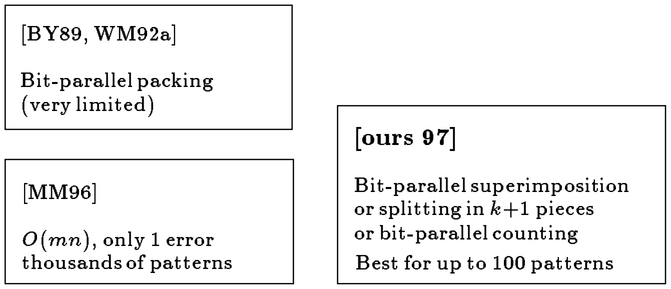

[BY89, WM92a]
Bit-parallel packing (very limited)
[MM96]
$O(mn)$, only 1 error thousands of patterns
[ours 97]
Bit-parallel superimposition or splitting in $k+1$ pieces or bit-parallel counting
Best for up to 100 patterns

Figure 3.6: Taxonomy of algorithms that search multiple patterns allowing errors.

- Information retrieval: when synonym or thesaurus expansion is done on a keyword and the text is error-prone, we may want to search all the variants allowing errors.
- Batched queries: if a system receives a number of queries to process, it may improve efficiency by searching all them in a single pass.
- Single-pattern queries: some algorithms for a single pattern allowing errors (e.g. pattern partitioning, see Section 5.5) reduce the problem to the search of many subpatterns allowing less errors, and they benefit from multipattern search algorithms.

The number $r$ of patterns may range from a few ones (in applications of synonym or thesaurus expansion) to thousands (in spelling applications). Of course the goal is to be more efficient than $r$ sequential searches.

The algorithm of Wu and Manber [WM92a] is able to search many patterns in parallel allowing errors. However, this capability is extremely limited since all the patterns must fit in a computer word, i.e. $rm \leq w$. If there are more patterns, many separate searches have to be performed. The possible speedup is limited by $\lfloor w/m \rfloor$, i.e. the search cannot be parallelized at all if $m > w/2$.

In 1996, Muth and Manber [MM96] presented an algorithm able to search thousands of patterns in parallel with essentially no degradation in the search cost, which is $O(mn)$. However, the technique is limited to $k = 1$. The idea is based on the fact that we can allow just one deletion in the pattern and/or in the text and the result is the same. A hash table stores all the alternatives of eliminating one letter from all patterns, and the text is traversed doing the same and searching the $(m-1)$-length substrings in the hash table.

As part of this thesis, we presented two new algorithms for multipattern approximate searching in 1997 [BYN97b]. The first algorithm is based on our bit-parallel algorithm (Section 3.1.4). Since we can put more than one letter in the arrows of the automaton (thanks to the mechanism of classes of characters), we "superimpose" many automata and perform a single search with the superimposed automaton. The search works as a filter, since by superimposing `"this"` and `"wait"` we can match `"wais"` with *zero* errors. We present a different hierarchical verification technique which works on the number of patterns superimposed rather than on the pattern pieces. We analyze how many patterns can be superimposed so that the filter works well. The result is the fastest algorithm for medium error level and a small number of patterns (say, less than 15).

The other algorithm presented in [BYN97b] extends the filter of the $(k+1)$ pieces (see Section 3.1.3), so that we search now $r(k+1)$ pieces. This algorithm is the fastest one for a moderate number of patterns (say, 100 or less) and low error levels.

Also in 1997, we presented another filter for multipattern approximate searching [Nav97a] (also part of this thesis). The filter is a multipattern version of the counting filter [JTU96] of Section 3.1.3 (which is simplified and studied more in depth in our paper). The algorithm combines the filter with bit-parallelism: the counters for many patterns are kept in a single computer word and updated in parallel. The algorithm was the fastest for for moderate error levels when it was created, but it is not anymore after hierarchical verification appeared.

We implement all the referenced algorithms in this thesis, since all them were the best at some point.

## 3.3 Indexed Searching

If the text is large and has to be searched frequently, even the fastest on-line algorithms are not practical, and preprocessing the text becomes necessary. Therefore, many indexing methods have been developed for exact string matching [WMB94]. However, only a few years ago, indexing text for approximate string matching was considered one of the main open problems in this area [WM92a, BY92]. Hence, the area is rather new.

We divide this presentation in two classes of indices, one of them able to solve the general problem (*sequence-retrieving indices*) and the other able to solve the restricted case of an index on natural language that retrieves whole words that match the pattern (not any sequence), which we call *word-retrieving indices.* This is briefly discussed in Sections 2.10 and 2.7. We made contributions in both areas (see Figure 3.7).

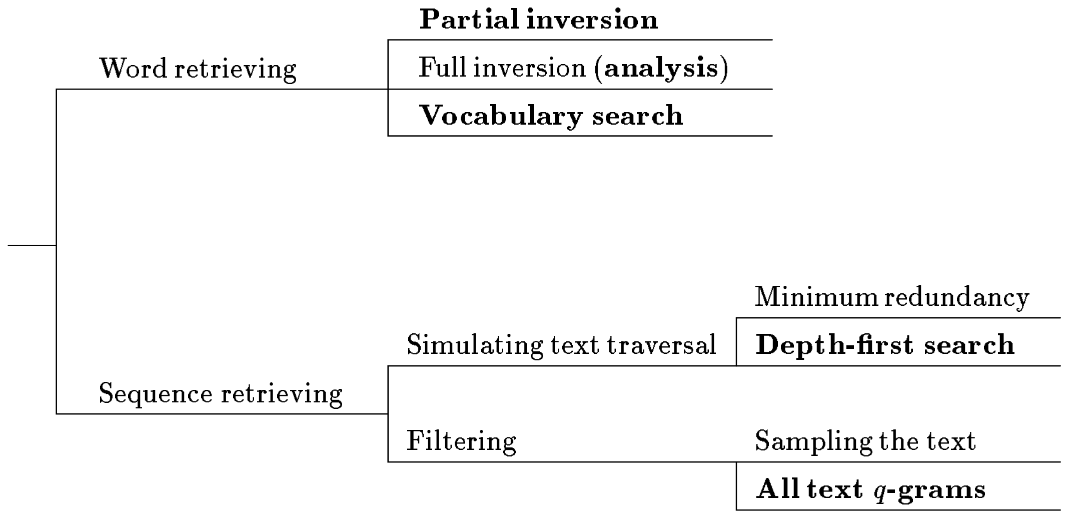


Figure 3.7: Taxonomy of indexed searching. The areas where we made contributions are in boldface.

When discussing indices, not only their retrieval performance is of interest, but also their space requirements and building cost. This last cost, however, is less interesting because it is assumed to be amortized over a number of searches. In general it is very difficult to give useful analytical results on the performance of sequence-retrieving indices. Only an experimental comparison can show their real performance.

Sequence-retrieving indices are much more immature than word-oriented ones (since they address a harder problem). Their speedups over the sequential algorithms are still unsatisfactory, and they take up too much space. Our goal in this thesis with respect to indexing has been to find alternatives which are more reasonable for typical text retrieval. We first cover word-retrieving indices and then the two different kinds of sequence-retrieving indices.

### 3.3.1 Word-Retrieving Indices

[See Figure 3.8]

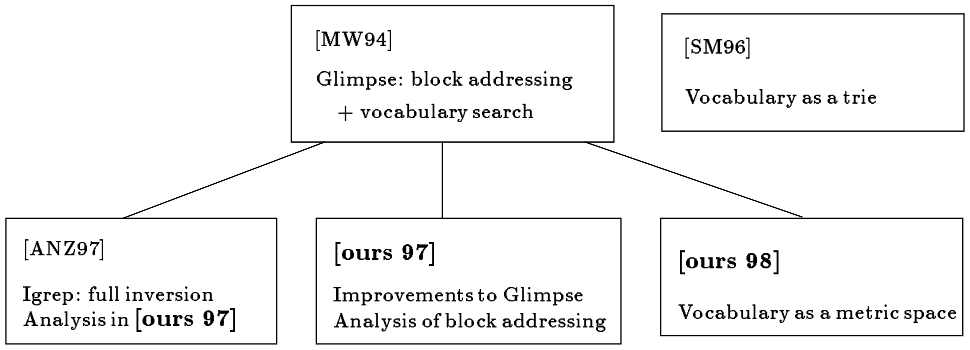


Figure 3.8: Taxonomy of word retrieving indices.

Word retrieving indices are basically inverted indices (see Section 2.10), where approximate searching is converted into a sequential search over the vocabulary. An inverted index can in principle be built in $O(n)$ time by keeping the vocabulary in a trie data structure and storing the list of occurrences at the leaves. However, this is not realistic if the index does not fit in main memory. There exist algorithms that work well on secondary memory whose cost is $O(n \log(n/M))$ where $M$ is the amount of available main memory. It is possible nowadays to build an inverted index for 1 Gb of text in 20 minutes or so. The space requirements of these indices is $O(n)$ if all the occurrences of the words are stored (typically 15%-40% of the text size if stopwords are not indexed). If block-addressing is used these requirements are smaller, as we see next.

The first proposal for a word-retrieving index (called *Glimpse*) was due to Manber and Wu in 1993 [MW94]. In a very practical approach, they propose a scheme based on a modified inverted file and sequential approximate search, as explained in Section 2.10. The text is logically divided into "blocks". The index stores all the different words of the text (the "vocabulary"). For each word, the list of the blocks where the word appears is kept.

To search a word allowing errors, an on-line approximate search algorithm (in this case, Agrep [WM92b]) is run over the vocabulary. Then, for every block where a matching word is present, a new sequential search is performed over that block (using Agrep again). The search in the vocabulary is cheap because it is small compared to the text size (Section 2.9).

The idea of using blocks makes the index small, at the cost of having to traverse parts of the text sequentially. The index is small not only because the pointers to the blocks are small, but also because all the occurrences in a single block are referenced only once.

Glimpse uses 250–256 blocks, which works well for moderate-size texts. For larger texts, it is possible to point to files instead of blocks, or even to occurrences of words (full inversion). Typical figures for the size of the index with respect to the text are: 2-4% for blocks, 10-15% for files, and 25-30% for words. Glimpse works well for texts of up to 200 Mb and moderate error ratio. Queries are answered in a few seconds, depending on the complexity of the query.

In 1997, as part of this thesis, we implemented a similar index and improved its search algorithm [BYN97a]. In the same lines of Glimpse, the text is divided into blocks and an on-line approximate search algorithm is run over the vocabulary (in this case, our bit-parallel algorithm [BYN96b], which is especially well suited for short patterns like words). However, once the list of matching words of each block is obtained, the approximate algorithm is not used anymore. Instead, a multiple *exact* pattern matching algorithm is used to search the matching words in the blocks. It is shown experimentally that this is up to five times faster than Glimpse.

In the same work, it is also shown analytically and experimentally that it is possible to have a sublinear-size index with sublinear search times, even for approximate word queries. A practical example shows that the index can be $O(n^{0.94})$ in space and in retrieval time. We also applied the analysis to Web document size statistics, obtaining new interesting results. This is a very important analytical result which is experimentally validated and makes a very good case for the practical use of this kind of index. Moreover, these indices are amenable to compression. Block-addressing indices can be reduced to 10% of their original size [BMNM+93], and the first works on searching the text blocks directly in their compressed form are just appearing [MNZBY98b, MNZBY98a] with very good performance in time *and* space (not part of this thesis).

Also in 1997, Araújo, Navarro and Ziviani take the approach of *full inversion* in an index called *Igrep* [ANZ97]. For each word, the list of all its occurrences in the text are kept and the text is never accessed. The search on the vocabulary is as before (using [BYN96b]), but the second phase of the search changes completely: once the matching words in the vocabulary are identified, all their lists are merged. Phrases can also be searched, by splitting them into words. The approach is much more resistant than Glimpse to the size of the text collection, and is shown to work well with text collections of more than 1 Gb. The price is the higher space requirements (30%-40% of the text size). It is easy to compress these indices to at least 15-20% of the text size by storing differential instead of absolute positions, although this makes impossible the use of some search optimization techniques. A 1 Gb text collection can be searched for single word queries in nearly 2 seconds for $k \leq 2$. Even using pointers to words, Glimpse does not work well with such large texts.

The analytical results of [ANZ97] are part of this thesis (not the rest). The analysis shows that, under the assumption that the vocabulary size is $O(n^{\beta})$ for $\beta \approx 0.5$ (which is validated in that work and in previous ones [Hea78]), the retrieval costs are near $O(n^{0.4..0.8})$ for useful searches (i.e. those with reasonable precision).

Notice that the part of the search which is still not optimized is the sequential search in the vocabulary. This is because the vocabulary is not very large (a few megabytes) and therefore can be sequentially searched in a few seconds. Although this solution is reasonable for a single end-user, other setups may require faster algorithms. In 1996, Shang and Merettal [SM96] used a trie to arrange all the words of a dictionary and reported improvements for $k = 1$. This is a particular case of what is done in the next section for the general problem. Although the algorithm is very fast, it poses high space requirements.

In 1997, as part of this thesis [BYN98b] we presented a different search scheme based on regarding the vocabulary as a *metric space* and using a data structure to index such metric space. A metric space is a set of elements with a distance function (see Section 2.1), which in our case is the set of words and the edit distance function. Using a suitable data structure to index the vocabulary as a metric space, the search time is sublinear and practical reductions of up to 60% in the search time are obtained. We need much less space than [SM96].

### 3.3.2 Simulating Text Traversal

[See Figure 3.9]

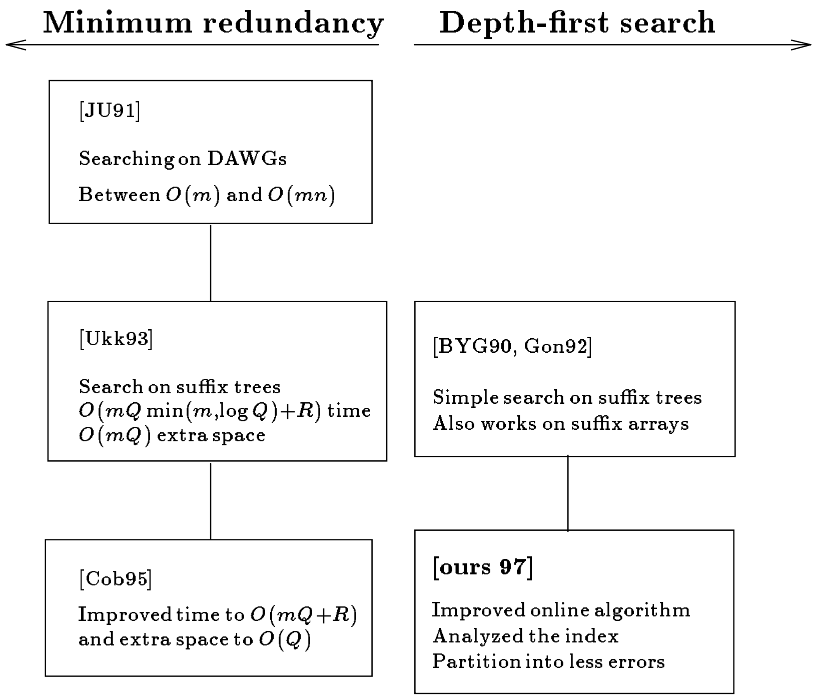

Minimum redundancy
Depth-first search
[JU91]
Searching on DAWGs
Between $O(m)$ and $O(mn)$
[Ukk93]
Search on suffix trees
$O(mQ\min(m,\log Q)+R)$ time
$O(mQ)$ extra space
[BYG90, Gon92]
Simple search on suffix trees
Also works on suffix arrays
[Cob95]
Improved time to $O(mQ+R)$
and extra space to $O(Q)$
[ours 97]
Improved online algorithm
Analyzed the index
Partition into less errors

Figure 3.9: Taxonomy of indices that simulate text traversal.

This type of sequence-retrieving index is based on simulating a sequential algorithm, but running it on the suffix tree or DAWG of the text (see Section 2.7) instead of the text itself. Since every different substring in the text is represented by a single node in the tree or automaton, it is possible to avoid the repetitions that occur in the text. Those indices take $O(n)$ space and construction time, but their construction is very inefficient if the text does not fit in main memory (their construction algorithm is not optimized for secondary memory). Moreover, they are very inefficient in space requirements, since they take at least 12 times the text size (i.e. an overhead of 1200% at least).

This can be partially overcome by using compression. In [KU96], a compression technique is proposed which obtains an index of size $O(nH/\log n)$, where $H$ is the entropy of the text characters, which in the worst case is $O(1)$. They show experiments on natural language where the space requirements are $2.5n$ bytes (i.e. 250% overhead). This is much better, although still insufficient for very large texts.

#### 3.3.2.1 Minimum Redundancy

The first work on this type of index is due to Jokinen and Ukkonen, in 1991 [JU91]. The proposal for this type of index is based on the DAWG of the text (see Section 2.7), where the idea is to traverse the DAWG instead of the text. Since each node of the DAWG represents a different substring of the text, this traversal avoids to process many times the same substring, therefore avoiding the text redundancies.

In 1993, Ukkonen presented an algorithm in the lines of [JU91], this time based on suffix trees instead of DAWGs [Ukk93]. The DAWG functionality was replaced with suffix links (see Section 2.7). This algorithm is based on the fact that the state of the search at a given point in the text is only influenced by the last characters read ($m+k$ or less). Ukkonen calls "viable prefixes" the substrings that can be prefixes of an approximate occurrence of the pattern. The algorithms traverse in the suffix tree all the different viable prefixes, simulating the dynamic programming algorithm behavior. The already visited states store their column values to avoid recomputation. Ukkonen presents three different algorithms, with times $O(mQ + n)$, $O(mQ \log Q + R)$ and $O(m^2 Q + R)$, where $R$ is the number of matches found and $Q$ is the number of viable prefixes, which is shown to be $O(\min(n, m^{k+1}\sigma^k))$. The extra space needed is $O(mQ)$.

In 1995, Cobbs [Cob95] used a very similar idea but with improved results: $O(mQ+R)$ search time and $O(Q)$ extra space.

#### 3.3.2.2 Depth-First Search

Around 1988, Gonnet proposed (independently of the above work) a simplified version of [Ukk93], to be run on suffix trees (later applied to computational biology in Darwin [Gon92]). The idea is a limited depth-first search on the suffix tree. Since every substring of the text (i.e. every potential occurrence) starts at the root of the suffix tree, it is sufficient to explore every path starting at the root, descending by every branch up to where it can be seen that that branch does not represent the beginning of an occurrence of the pattern. This algorithm inspects more nodes than [Ukk93], but it is simpler and does not need the suffix links. For instance, with an additional $O(\log n)$ time factor penalty, the algorithm runs on suffix arrays, which take 4 times the text size instead of 12 times. This scheme is analyzed in [BYG90], where it is shown that at most $O((\sigma - 1)^{k(m-k-1)})$ nodes are inspected[2].

As part of this thesis [BYNST97] we implemented the algorithm [BYG90, Gon92] over suffix trees in 1997, replacing dynamic programming with our bit-parallel algorithm (explained in Section 3.1.4). This algorithm cannot work on the more complex setup of [Ukk92], since that one needs some adaptations of the dynamic programming algorithm that were not easy to parallelize. We also analyze this algorithm, finding that the number of inspected nodes is sublinear for $\alpha < (1 - e/\sqrt{\sigma})(\log_\sigma n)/m$ if $\log_\sigma n \leq m$, and for $\alpha < (\log_\sigma n)/m - 1$ or $\alpha < 1 - e/\sqrt{\sigma}$ otherwise. This algorithm can also be run on a suffix array, i.e. on an index which is 4 times the text size. The experimental results shows that this idea is up to 150 times faster than Cobbs' algorithm, and therefore the simplification pays off.

In the same work, we adapt the on-line pattern partitioning techniques of [BYN96b] (see Section 3.1.3), so that the pattern is split in $j$ subpatterns which can be searched with the bit-parallel algorithm with at most $k/j$ errors. All the occurrences are collected and the matching positions verified. This idea worked surprisingly well, being an order of magnitude faster than all existing indexing schemes. We suggest how to reduce the space usage for this index and show that more study is needed to fully understand how this idea works. The suggested index is in fact based on text samples but uses approximate searching on them.

[2]In fact, it is easy to show that at most $O(\sigma^{m+k})$ nodes are inspected, but the result of [BYG90] holds for general acyclic regular expressions.

### 3.3.3 Filtration Indices

[See Figure 3.10]

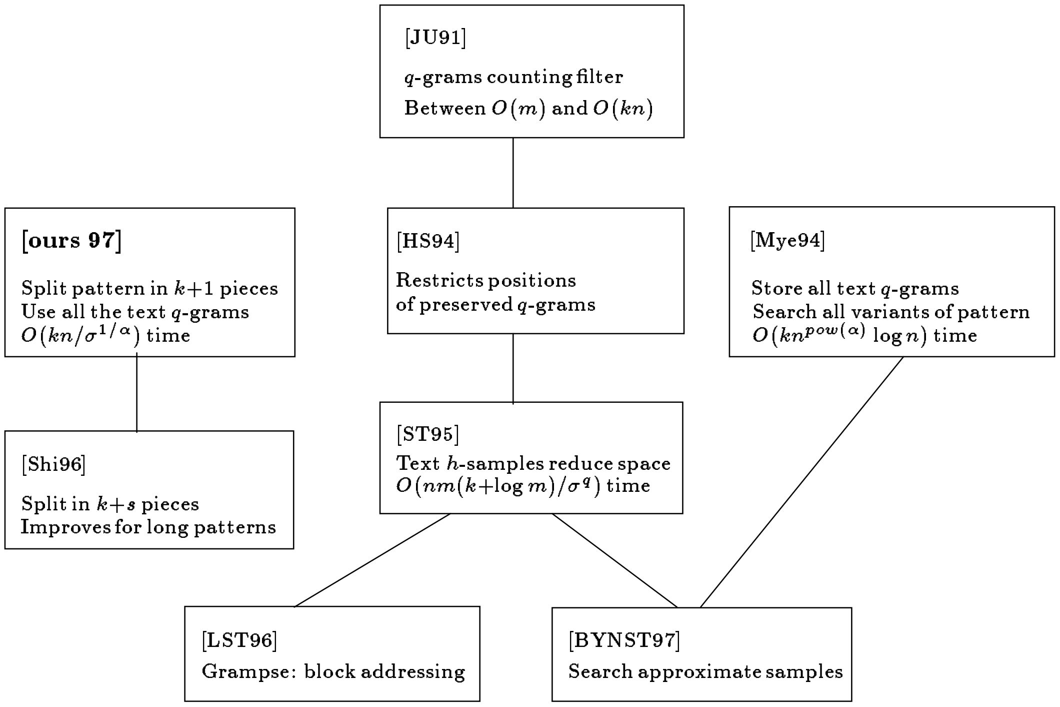

[JU91]
$q$-grams counting filter
Between $O(m)$ and $O(kn)$
[ours 97]
Split pattern in $k+1$ pieces
Use all the text $q$-grams
$O(kn/\sigma^{1/\alpha})$ time
[HS94]
Restricts positions of preserved $q$-grams
[Mye94]
Store all text $q$-grams
Search all variants of pattern
$O(kn^{pow(\alpha)} \log n)$ time
[Shi96]
Split in $k+s$ pieces
Improves for long patterns
[ST95]
Text $h$-samples reduce space
$O(nm(k+\log m)/\sigma^q)$ time
[LST96]
Grampse: block addressing
[BYNST97]
Search approximate samples

Figure 3.10: Taxonomy of filtration indices.

This second type of sequence-retrieving indices is based on adapting an on-line filtering algorithm (see Section 3.1.3). The filters that are based in locating substrings of the patterns without errors can build an index to quickly find those substrings. The advantage of those indices is that they are smaller. They can also be built in linear time (with similar algorithms as for inverted indices). See Section 2.10 for a quick initial discussion.

Their size and performance depend a lot on the length of the $q$-grams stored. If the $q$-grams are reasonable short and pose no space problems, the main problem of these indices is their linear space consumption to store the $q$-grams positions in the text (this can vary depending on how many text $q$-grams are indexed). However, it is possible to reduce the space consumption, as shown in [KS96]. The method makes use of the Lempel-Ziv idea for text compression which replaces each repetition of the text by a reference to its first occurrence. The paper shows how to efficiently search on such compressed data structure, which takes $O(n/\log n)$ space if $q$ is small.

Two crucial factors that affect the size of the index are the length of the $q$-grams (i.e. the value $q$), and the *sampling step* $h$, which is the distance among two text samples. The size of the "vocabulary" (i.e. number of different $q$-grams) depends exponentially on $q$ (it is upper bounded by $\sigma^q$), and the number of pointers to the text is $n/h$. The value $q$ is normally rather small (3 to 5 in practice), and it cannot be shorter without affecting the filtration efficiency. With respect to the sampling step, we can divide the $q$-gram indices in two types: those that take all the $(m-q+1)$ overlapping $q$-grams of the pattern and only some samples of the text ($n/h$ samples), and those that take non-overlapping $q$-grams in the pattern (we say that they "sample" the pattern) and all the $(n-q+1)$ $q$-grams of the text. The second type obviously takes more space, but is able to cope with higher error ratios. In fact, the text sampling indices seem to tolerate only low error levels for practical natural language applications.

The idea of using $q$-grams for indexed approximate string matching is very old [Ull77, OM88]. However, the methods were not fully developed until the 1991 work of Jokinen and Ukkonen [JU91]. Their proposal for this case is an index of text $q$-grams, based on the observation that if an approximate match of $P$ with at most $k$ differences ends at $T...j$, then at least $(m+1-(k+1)q)$ $q$-grams of $P$ occur in $T_{j-m+1..j}$. This is because $T_{j-m+1..j}$ includes $m-q+1$ $q$-grams, of which at most $kq$ can be altered by $k$ edit operations. The text is divided into two layers of consecutive, non-overlapping blocks of length $2(m-1)$. Then, the number of pattern $q$-grams in each block is counted. For each block with enough pattern $q$-grams, the respective text area is examined using dynamic programming. Notice that this is in fact an off-line version of the filter [Ukk92]. Since it is based on counting, we do not classify this index as sampling the text or not, although all the indices that followed can be divided in this manner.

#### 3.3.3.1 All $q$-grams on the Text

In 1994, Myers [Mye94] proposed an index where every sequence of the text up to a given length $q$ is stored, together with the list of its positions in the text. To search for a pattern of length at most $q-k$, all the maximal strings with edit distance at most $k$ to the pattern are generated, and each one is searched. Later, the lists are merged. Longer patterns are split in as many pieces as necessary to make them of the required length. Instead of simply verifying each of the approximate occurrences of the substrings, the algorithm goes up level by level in the partition process, obtaining the approximate occurrences of that level by combining those of their two children in the next level. Query complexity is shown to be $O(kn^{pow(\alpha)}\log n)$ on average, where $pow(\alpha)$ is a concave function of $\alpha$ satisfying $pow(0)=0$. This is sublinear when $pow(\alpha)<1$, which restricts the error ratios up to where the scheme is efficient. This maximum allowed error ratio increases with the alphabet size. For example, the formula shows that $\alpha_{\max}$ is 0.33 for $\sigma=4$ and 0.56 for $\sigma=20$. Experiments confirm those computations.

In 1994, Holsti and Sutinen [HS94] improved the filtration condition of [JU91], using the fact that a preserved pattern $q$-gram cannot move more than $k$ positions away from its original position. This method does not use the block-oriented scheme of Jokinen and Ukkonen, but a window-oriented approach: each occurrence of a pattern $q$-gram in $T$ marks a corresponding window where an approximate match might be located. The search times slightly improve those of [JU91], but the index keeps the same: all the text $q$-grams are stored.

In 1996, Shi [Shi96] proposed another alternative based on storing all the text $q$-grams (see also Section 3.1.3). The pattern is split in $k+s$ parts, which cover the whole pattern and do not overlap. The paper reports that much less verifications are triggered than for the particular case $s=1$. However, this is true for quite long patterns, which are very rare in typical text retrieval. Shi proposes the use of a suffix tree to locate the occurrences of the pieces of the pattern, but a much smaller $q$-gram index could be used.

In 1998 and as part of this thesis [BYN97c] we implemented a simple and practical index, based on splitting the pattern in $k+1$ pieces (see Section 3.1.3). The goal is to obtain an index which is useful for natural language, as the on-line version of this filter is the most efficient for short patterns like words. All the $n$ text $q$-grams are indexed. To search a pattern, it is split in $k+1$ pieces, each piece is searched with no errors in an index, and all their occurrences are verified for a complete match. Since in natural language a simple equal-length partition may give very bad results depending on the resulting substrings, an $O(m^2k)$ dynamic programming algorithm is used to select the best pattern split, i.e. the one that minimizes the total number of text positions to verify. It is shown that the search time is $O(kn/\sigma^{1/\alpha})$. For instance, with an index which is 1.5 to 3 times the text size, reasonable approximate queries take 20% to 60% of the time of the on-line algorithm. We found experimentally that in typical text searching it is impossible in practice to use block addressing and still have a useful index. We believe that this is currently the most practical index for typical text retrieval. In our experimental results we show that this index is many times faster than Myers' on English text, although it is slower on DNA. It is also slower than those based on suffix trees, but these pose impractical space requirements.

#### 3.3.3.2 Sampling the Text

Sutinen and Tarhio gave in 1995 [ST95] a static version of their on-line algorithm (see Section 3.1.3). The idea is based on observing a sequence of $q$-samples, i.e. non-overlapping $q$-grams of the text at fixed periods of length $h$ ($h$ is determined by the error level to tolerate). The main contribution of the static variation is saving space: only every $h$-th $q$-sample of text $T$ is stored into the index. At first sight, this seems to result in a different index for each $m$ and $k$, but they can adjust $s$ in the formula of $h$ so that the index, precomputed according to a fixed $h$, can be applied. They show that their method needs less space for $\alpha < 0.3$, and is effective for low error levels (e.g. $m = \sigma = 40$ and $k \leq 4$). The search times are $O(nm(k + \log m)/\sigma^q)$. Compared to our index [BYN97b], this index takes much less space, though it much less tolerant to errors as well (in fact, too restrictive for natural language, although it improves for longer patterns). This can be noticed by comparing the respective on-line algorithms. Direct experimental results, however, are still not possible because the current implementation does not support approximate searching.

The block-addressing idea presented in Section 2.10 to reduce space requirements can be applied to this type of index too, at the cost of (much) higher search time. This has been presented in 1996 in a variation of Glimpse, called Grampse [LST96]. In addition to conventional $q$-grams, Grampse uses also "gapped $q$-grams" (i.e. their letters are not consecutive in the pattern but picket at fixed intervals). Compared to Glimpse, Grampse works faster in locating sequences of words, and is useful for agglutinating languages such as Finnish.

In 1997, the possibility of an index based on approximate occurrences of parts of the pattern (instead of exact occurrences) was proposed in [BYNST97] (not part of this thesis). Similarly to the approach of [Mye94], all the elements at a given distance of the $q$-grams of the pattern are searched in the index. The filtration condition, however, is stricter since it requires that more than one element is present in the potential match. This index has not been implemented up to date.

# Chapter 4

# Basic Tools

In this chapter we present a number of basic tools which are used throughout the thesis, and therefore we have decided to collect them in a single chapter previous to the rest of the work. These results have been published in [BYN96b, BYN96a, BYN98d, NBY98b, BYN98e].

## 4.1 Statistics of the Problem

In this section we (partially) answer some natural questions about the distribution and statistical behavior of approximate string matching, which turn out to be very hard. These questions are not only interesting by themselves, but are also essential for the average-case analysis of all the algorithms of this thesis and for the practical selection of the correct algorithms for each scenario. Some of these questions are: what is the probability of an approximate occurrence? How many occurrences are there on average? How many columns of the dynamic programming matrix are active? We give here our new results.

In all the average-case analysis of this work we assume that the patterns are not extended. An easy way to consider classes of characters is to replace $\sigma$ by $\sigma/s$ in all the formulas, where $s$ is the size of the $P_i$ sets corresponding to pattern positions. This is because the probability of matching such pattern position is not $1/\sigma$ anymore, but $s/\sigma$. For other extended patterns our analysis simply do not apply, and a case-by-case consideration is necessary.

An analysis of the probability of the occurrence of a pattern allowing errors can be found in [RS97]. However, the final expression is extremely complex, has $P$ and $k$ built-in, and does not allow deriving any general result. It can be used for, given a *fixed* $P$ and $k$, compute the probability (using a computer program) for that particular pattern and number of errors. It is of no use to derive general results. The results that we present in this chapter are simpler and of general use[1].

### 4.1.1 Probability of Matching

Let $f(m, k)$ be the probability of a random pattern of length $m$ matching a given text position with $k$ errors or less (i.e. that the text position is reported as the end of a match). We find analytical upper and lower bounds for this probability and present later an experimental verification.

[1]We have also developed a simple C+Maple program which, given a specific pattern $P$ and a number of errors $k$, builts its NFA, converts it to a DFA and then uses the Markov model to compute the probability of the final states, therefore finding its exact matching probability as a function of $\sigma$. But this is not the real problem.

In particular, we are interested in which is the error level up to where the probability of verifying is exponentially decreasing with $m$. The importance of being exponentially decreasing with $m$ is that the cost of verifying a text position is $O(m^2)$, and therefore if that event occurs with probability $O(\gamma^m)$ for some $\gamma < 1$ then the total cost of verifications is $O(m^2\gamma^m) = o(1)$, which makes the verifications cost negligible. On the other hand, we show that as soon as the cost ceases to be exponentially decreasing it begins to be at least $1/m$, which yields a total verification cost of $O(mn)$. This is the same cost of plain dynamic programming. Hence, the limit $\alpha < 1 - e/\sqrt{\sigma}$ which we prove next corresponds, in theory, to the maximum error level up to where *any* algorithm based on filtration can work well[2].

#### 4.1.1.1 An Upper Bound

We show that the matching probability is $O(\gamma^m)$ for

$$\gamma = \left(\frac{1}{\sigma\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)^2}\right)^{1-\alpha} \quad \leq \quad \left(\frac{e^2}{\sigma(1-\alpha)^2}\right)^{1-\alpha} \tag{4.1}$$

If our only aim is to make such probability exponentially small with $m$, we take the bound for $\gamma = 1$ and consider valid any error level *strictly* smaller than the bound. This is

$$\alpha \quad < \quad \alpha^* \quad = \quad 1 - \frac{e}{\sqrt{\sigma}} \tag{4.2}$$

while given a $\gamma$ value the condition on $\alpha$ is given by Eq. (4.4), which is shown later.

To prove $f(m,k) = O(\gamma^m)$, we consider an upper bound to $f$: suppose a text area $T_{a..b}$ matches the pattern. Since we only report segments whose last character matches the pattern, we know that $T_b$ is in $P$. We consider $T_a$ as the first character matching the pattern. Then, the length $s = b - a + 1$ is in the range $m-k..m+k$. Since there are up to $k$ errors, at least $m-k$ characters of the pattern must be also in the text. Under a uniform model, the probability of that many matches is $1/\sigma^{m-k}$. Since these characters can be anywhere in the pattern and in the text, we have

$$f(m,k) \leq \sum_{s=m-k}^{m} \frac{1}{\sigma^{m-k}} \binom{m}{m-k} \binom{s-2}{m-k-2} + \sum_{s=m+1}^{m+k} \frac{1}{\sigma^{s-k}} \binom{m}{s-k} \binom{s-2}{s-k-2} \tag{4.3}$$

where the two combinatorials count the ways to choose the $m-k$ (or $s-k$) matching characters from the pattern and from the text, respectively. The "$-2$" in the second combinatorials are because the first and last characters of the text must match the pattern. We divided the sum in two parts because if the area has length $s > m$, then more than $m-k$ characters must match, namely $s-k$. See Figure 4.1.

First assume constant $\alpha$ (we cover the other cases later). We begin with the first summation, which is easy to solve exactly to get $(1-\alpha)\binom{m}{k}^2/\sigma^{m-k}$. However, we prefer to analyze its largest term (the last one), since it is useful for the second summation too. The last term is

$$\frac{1}{\sigma^{m-k}} \binom{m}{m-k} \binom{m-2}{m-k-2} \quad = \quad \frac{(1-\alpha)^2}{\sigma^{m-k}} \binom{m}{k}^2 \left(1 + O\left(\frac{1}{m}\right)\right)$$

[2]Notice that this limit is useless for $\sigma < 8$, but is the limit we can formally prove. We show shortly that replacing $e$ by 1.09 fits better the real data.

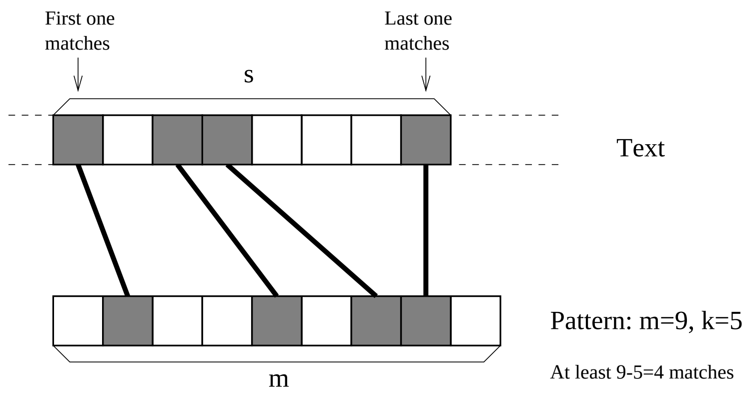


Figure 4.1: Upper bound for $f(m,k)$.

and by using Stirling's approximation $x! = (x/e)^x\sqrt{2\pi x}(1+O(1/x))$ we have

$$\frac{(1-\alpha)^2}{\sigma^{m-k}}\left(\frac{m^m\sqrt{2\pi m}}{k^k(m-k)^{m-k}\sqrt{2\pi k}\sqrt{2\pi(m-k)}}\right)^2\left(1+O\left(\frac{1}{m}\right)\right)$$

which is

$$\left(\frac{1}{\sigma^{1-\alpha}\alpha^{2\alpha}(1-\alpha)^{2(1-\alpha)}}\right)^m \; m^{-1}\;\left(\frac{1-\alpha}{2\pi\alpha}+O\left(\frac{1}{m}\right)\right)$$

where the last step is done using Stirling's approximation to the factorial.

Clearly, for the summation to be $O(\gamma^m)$ $(\gamma<1)$, this largest term must be of that order, and this happens if and only if the base of the exponential is $\leq\gamma$. On the other hand, the first summation is bounded by $k+1$ times the last term, so the first summation is $O(\gamma^m)$ if and only if this last term is (recall that our exponential is multiplied by $m^{-1}$ and therefore we can safely multiply it by $k+1$). That is

$$\sigma \;\geq\; \left(\frac{1}{\gamma\alpha^{2\alpha}(1-\alpha)^{2(1-\alpha)}}\right)^{\frac{1}{1-\alpha}} \;=\; \frac{1}{\gamma^{\frac{1}{1-\alpha}}\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)^2}$$

It is easy to show analytically that $e^{-1}\leq\alpha^{\frac{\alpha}{1-\alpha}}\leq 1$ if $0\leq\alpha\leq 1$, so for $\gamma=1$ it suffices that $\sigma\geq e^2/(1-\alpha)^2$, or equivalently $\alpha\leq 1-e/\sqrt{\sigma}$ (hence Eq. (4.2)), while for arbitrary $\gamma$,

$$\alpha \;\leq\; 1-\frac{e}{\sqrt{\sigma}\gamma^{\frac{1}{2(1-\alpha)}}} \tag{4.4}$$

is a sufficient condition for the largest (last) term to be $O(\gamma^m)$, as well as the whole first summation.

We address now the second summation, which is more complicated. First, observe that

$$\sum_{s=m+1}^{m+k}\frac{1}{\sigma^{s-k}}\binom{m}{s-k}\binom{s-2}{s-k-2} \;\leq\; \sum_{s=m}^{m+k}\frac{1}{\sigma^{s-k}}\binom{m}{s-k}\binom{s}{k}$$

a bound that we later find tight. In this case, it is not clear which is the largest term. We can see each term as

$$\frac{1}{\sigma^r}\binom{m}{r}\binom{k+r}{k}$$

where $m-k \leq r \leq m$. By considering $r = xm$ ($x \in [1-\alpha, 1]$) and applying again Stirling's approximation, the problem is to maximize the base of the resulting exponential, which is

$$h(x) = \frac{(x+\alpha)^{x+\alpha}}{\sigma^x x^{2x} (1-x)^{1-x} \alpha^\alpha}$$

Elementary calculus leads to solve a second-degree equation that has roots in the interval $[1-\alpha, 1]$ only if $\sigma \leq \alpha/(1-\alpha)^2$ (indeed, this condition is necessary just for $x \geq 1-\alpha$). Since due to Eq. (4.4) we are only interested in $\sigma \geq 1/(1-\alpha)^2$, $\delta h(x)/\delta x$ does not have roots, and the maximum of $h(x)$ is at $x = 1-\alpha$. That means $r = m-k$, i.e. the first term of the second summation, which is the same largest term of the first summation.

We conclude that

$$f(m,k) \;\leq\; \frac{2k+1}{m}\,\gamma^m \left(1 + O\left(\frac{1}{m}\right)\right) \;=\; O\left(\gamma^m\right)$$

and therefore Eqs. (4.2) and (4.4) hold for the whole summation. When $\gamma$ reaches 1, the probability is very high, since only considering the term $s = m$ we have $\Omega(1/m)$.

Since we obtained an $O()$ result, it suffices for the condition to hold after a given $m_0$, so if $k = o(m)$ we always satisfy the condition.

#### 4.1.1.2 A Lower Bound

On the other hand, the only optimistic bound we can prove is based on considering that only replacements are allowed (i.e. Hamming distance). In this case, given a pattern of length $m$, the number of strings that are at distance $i$ from it are obtained by considering that we can freely determine the $i$ places of mismatch, and that at those places we can put any character except that of the pattern, i.e.

$$\binom{m}{i}(\sigma-1)^i \;=\; \binom{m}{i}\,\sigma^i\,\left(1 + O(1/\sigma)\right)$$

Although we should sum the above probabilities for $i$ from zero to $k$, we use the largest $i = k$ as a (tight) lower bound. Hence, the probability of matching is obtained by dividing the above formula (with $i = k$) by $\sigma^m$ (the total number of possible text windows of length $m$), to obtain

$$f(m,k) \geq \binom{m}{k}\frac{1}{\sigma^{m-k}} \;=\; \frac{m^m}{k^k (m-k)^{m-k} \sigma^{m-k} \sqrt{m}}\,\Theta(1) \;=\; \left(\frac{1}{\alpha^{\frac{\alpha}{1-\alpha}}(1-\alpha)\sigma}\right)^{(1-\alpha)m} \Theta(m^{-1/2})$$

which since $e^{-1} \leq \alpha^{\frac{\alpha}{1-\alpha}} \leq 1$, can be lower bounded by $f(m,k) \geq \delta^m\; m^{-1/2}$, where

$$\delta = \left(\frac{1}{(1-\alpha)\sigma}\right)^{1-\alpha}$$

Therefore an upper bound for the maximum allowed value for $\alpha$ is $\alpha \leq 1 - 1/\sigma$, since otherwise we can prove that $f(m,k)$ is *not* exponentially decreasing on $m$ (i.e. it is $\Omega(m^{-1/2})$).

#### 4.1.1.3 Experimental Verification

We verify our analysis experimentally in this section. The experiment consists of generating a large random text ($n = 10$ Mb) and running the search of a random pattern on that text, allowing $k = m$

errors. At each text character, we record the minimum allowed error $k$ for which that text position would match the pattern. We repeat the experiment with 1,000 random patterns.

Finally, we build the (cumulative) histogram, finding how many text positions have matched for each $k$ value. We consider that $k$ is "safe" up to where the histogram values become significant, that is, as long as few text positions have matched. The threshold is set to $n/m^2$, since $m^2$ is the cost of verifying a match. However, the selection of this threshold is not very important, since the histogram is extremely concentrated. For example, for $m$ in the hundreds, it moves from almost zero to almost $n$ in just five or six increments of $k$.

Figure 4.2 shows the results for $m = 300$, showing the maximum "safe" $\alpha$ value. The curve $\alpha = 1 - 1/\sqrt{\sigma}$ is included to show its closeness to the experimental data. Least squares give the approximation $\alpha^* = 1 - 1.09/\sqrt{\sigma}$, with a relative error smaller than 1%.

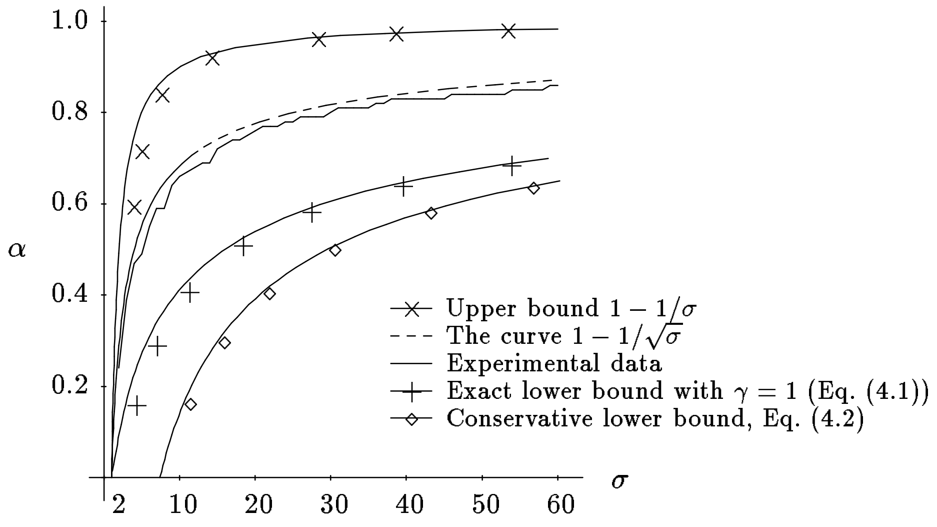


Figure 4.2: Theoretical and practical bounds for $\alpha$, for $m = 300$ and different $\sigma$ values. We plot the maximum $\alpha$ value which does not trigger too many verifications.

Figure 4.3 validates other theoretical assumptions. On the left we show that the matching probability undergoes a sharp increase at $\alpha^*$ (this is the histogram we have built). On the right we show that $\alpha^*$ is essentially independent on $m$. Notice, however, that our assumptions are a bit optimistic since for short patterns the matching probability is somewhat higher.

This shows that our upper bound analysis matches reality better, provided we replace $e$ by 1.09 in the formulas. We do so at the places of this work that require practical tuning, while we keep using $e$ (i.e. the value we can actually prove) in the theoretical developments.

### 4.1.2 Active Columns

In Section 3.1.1 we explained the cut-off heuristic of Ukkonen [Ukk85b], which for each character works a time proportional to the number of *active columns* in the dynamic programming matrix. At the end of each iteration the last active column may increase in one (if a horizontal automaton arrow is crossed from the last active column to the next one), or may decrease in one or more (if the last active column runs out of active states, the next-to-last may be well before it). In this case the algorithm goes backward in the matrix looking at the new last active column.

As we later use this same idea, it is interesting to know which is the average number of active

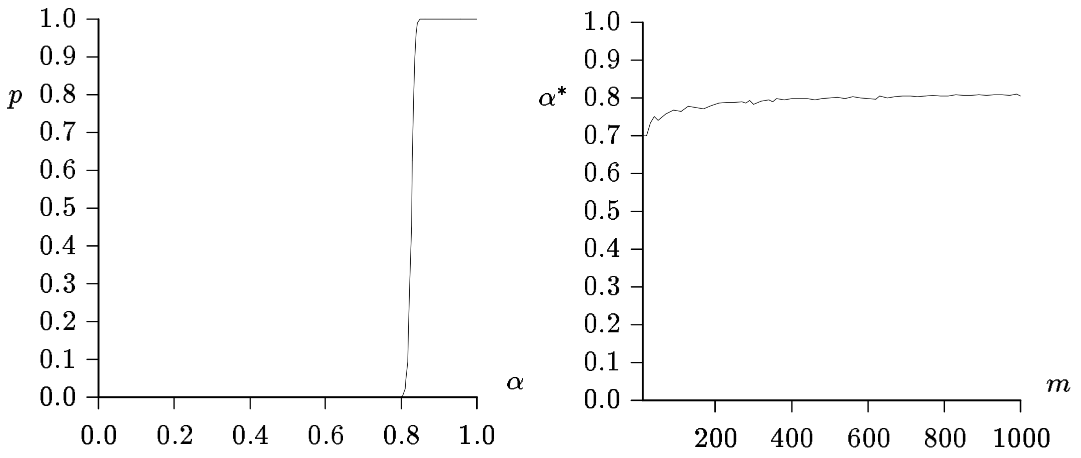


Figure 4.3: On the left, probability of an approximate match as a function of the error level ($m = 300$). On the right, maximum allowed error level as a function of the pattern length. Both cases correspond to random text with $\sigma = 32$.

columns. Ukkonen conjectured that the last active column was $O(k)$ on average and therefore his algorithm was $O(kn)$ on average. However, this was proved much later by Chang and Lampe [CL92]. We find here a tighter bound.

More formally, if we call $c_r$ the minimum row of an active state at column $r$ in the NFA of Section 2.2) [3], then the question is: which is the largest $r$ satisfying $c_r \le k$? The columns satisfying $c_r \le k$ are those *active*. We follow here the proof of [CL92] and find a tighter bound. If we call $L$ the last active column, we have

$$E(L) \;\le\; K + \sum_{r>K} r\ Pr(c_r \le k)$$

for any $K$. Since we know from the previous section that if $k/r < 1-e/\sqrt{\sigma}$, then $Pr(c_r \le k) = O(\gamma^r)$ with $\gamma < 1$, we take $K = k/(1-e/\sqrt{\sigma})$ to obtain

$$E(L) \;\le\; \frac{k}{1-e/\sqrt{\sigma}} + \sum_{r>k/(1-e/\sqrt{\sigma})} r\ O(\gamma^r) \;=\; \frac{k}{1-e/\sqrt{\sigma}} + O(1) \qquad (4.5)$$

which shows that, on average, the last active column is $O(k)$. This refines the proof of [CL92], which shows that the heuristic of [Ukk85b] is $O(kn)$.

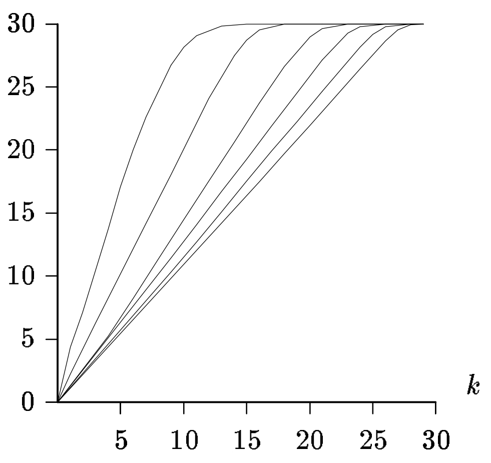
The $e$ of the above formula has the same source as in the previous section and hence can be replaced by 1.09 in practice. By using least squares on experimental data we find that a very accurate formula is

$$E(L) \;=\; 0.9\frac{k}{1-1.09/\sqrt{\sigma}} \qquad (4.6)$$

with a relative error smaller than 3.5%.

Figure 4.4 (left side) shows the last active column for random patterns of length 30 on random text, for different values of $\sigma$. Given the strong linearity, we take a fixed $k = 5$ and use least squares to find the slope of the curves. From that we obtain the 0.9 above. The right side of the figure shows the experimental data and the fitted curve (for $m = 30$). The results are the same for any $k$ less than $m(1-1.09/\sqrt{\sigma})$.

[3]Alternatively, $c_r$ is the value at row $r$ of the current column in the dynamic programming matrix (see Section



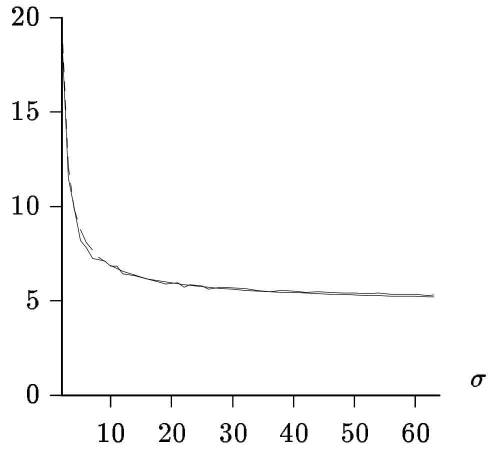


Figure 4.4: On the left, last active column for $\sigma$ = 2, 4, 8, 16, 32 and 64 (curves read from left to right). On the right, last active column for $k = 5$, experimental (full line) and theoretical (dashed).

## 4.2 Partitioning Lemmas

As explained in Section 3.1.3, it is well known that the problem of approximate string matching can be reduced to a number of subproblems by partitioning the pattern into subpatterns, which can be searched with less errors.

The oldest partitioning idea of this kind traces back to [WM92a], where they show that if the pattern is split in $k + 1$ pieces, then at least one piece must appear unaltered in an approximate occurrence of the pattern. This is obvious, since $k$ errors cannot alter the $k + 1$ pieces[4]. The idea is to search for all the $k + 1$ pieces at the same time, and each time a piece is found, the neighborhood of the match is checked for an approximate occurrence of the complete pattern. Figure 4.5 illustrates this idea.

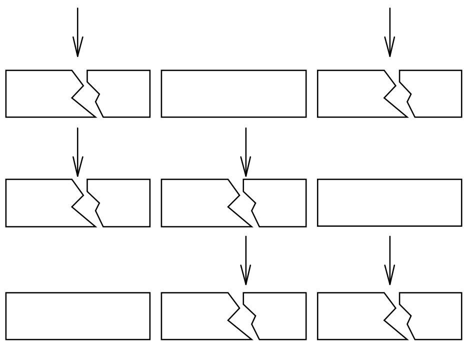


Figure 4.5: Illustration of the partitioning lemma for $k + 1$ pieces. In this case $k = 2$.

In [Mye94], a more general version of this lemma is presented. It is shown that the pattern can be cut in $2^i$ pieces, and that each piece can be searched with $\lfloor k/2^i \rfloor$ errors. It is also shown how to handle uneven partitions in this binary decomposition process.

In this section we present the most general version we have used of the pattern partitioning idea.

2.2).

[4]Notice that this "obvious" fact depends on the types of operations allowed. For instance, a single transposition can alter two pieces. However, the fact is true for our insertion, deletion and replacement operations.

This idea is used in many occasions later in this work.

**Partitioning Lemma:** If $Occ = T_{a..b}$ matches $P$ with $k$ errors, and $P = x_0\ P_1\ x_1\ ...\ x_{j-1}\ P_j\ x_j$ (i.e. a concatenation of subpatterns $P_i$ not necessarily contiguous but non-overlapping), then $Occ$ includes a substring that matches at least one of the $P_i$'s, with $\lfloor a_i k/A \rfloor$ errors, where $A = \sum_{i=1}^{j} a_i$.

**Proof:** Otherwise, each $P_i$ matches inside $Occ$ with at least $\lfloor a_i k/A \rfloor + 1 > a_i k/A$ errors. Summing up the errors of all the pieces we have more than $Ak/A = k$ errors and therefore a complete match is not possible.

The original partitioning lemma of [WM92a] corresponds to the case $j = k+1$, $a_i = 1$, and $P_i$'s contiguous and of similar length. That is, some piece is found with 0 errors. [Mye94] is another particular case.

In Chapter 5 we use a version of the Lemma with $j$ contiguous subpatterns which are found with $\lfloor k/j \rfloor$ errors, i.e. $a_i = 1$. In the next section we show a hierarchical verification technique that makes full use of the power of the $a_i$'s of the Lemma, but the patterns are still contiguous (i.e. $x_r = \varepsilon$), and is used in Chapter 5 and Section 6.1. Finally, a splitting optimization technique presented in Section 9.1 uses non-contiguous subpatterns (although the restricted case $j = k+1$).

Notice that, unless we are limited for some reason, the best we can do is to select contiguous subpatterns, since this reduces the probability of finding them in the text.

Our general version does not include a generalization of [WM92a] presented in [Shi96]: if we partition the pattern in $k+s$ pieces, then $s$ pieces appear unaltered. The idea is to force more than one piece to match in order to trigger a verification.

Yet another possible generalization which is not covered here is to allow that the subpatterns overlap. This is the basis of the $q$-gram ideas presented in [ST95, ST96] and others (see Sections 3.1.3 and 3.3.3). Overlapping pieces is in principle a bad idea because a single error may destroy many pieces. However, it leads to smaller indices and is useful for very low error levels or long patterns.

## 4.3 Hierarchical Verification

This is a technique we use in various forms throughout this work, aimed at minimizing the verification cost for potential occurrences. We present and analyze them now, and give more details later when they are used.

### 4.3.1 Pattern Splitting

In the previous section we have mentioned that the problem of approximate string matching can be decomposed in $j$ searches with $\lfloor k/j \rfloor$ errors, and that the matches found by any of those $j$ searches are to be checked for an occurrence of the complete pattern. The Lemma is useful if we have an efficient algorithm for the smaller subproblems, so that we use the less efficient algorithm (e.g. dynamic programming) only in the neighborhood of the matches of a subpattern. In this section we consider the best way to perform the verification of the candidate areas.

A first alternative is just to run the more expensive algorithm in the candidate areas. If one is able to determine which piece of the pattern was found, then it suffices to check an area of length $m+2k$. This area is obtained by placing the pattern in the text so that the piece found is aligned to the text position where it was found, and allowing $k$ more characters at each end of that area. This is because it is useless consider more than $k$ characters at each end, since if more than $k$ insertions are necessary a match is not possible. On the other hand, if the pattern position of the piece found is not known, we must consider $m + k$ characters in each direction from the matching text position, for a total area of size $2(m + k)$. In any case, if an $O(mn)$ verification algorithm is used, the cost of verifying each candidate area is $O(m^2)$.

In this section we consider a more elaborate verification algorithm, which we call "hierarchical verification". It has been mentioned (in a different context) in [Mye94], although our version is simpler and better suited to text retrieval.

First assume that $j$ is a power of 2. Then, we form a tree by recursively splitting the pattern in two halves of size $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$ (halving also the number of errors, i.e. $\lfloor k/2 \rfloor$) until the pieces are small enough to be searched with the core algorithm. Those pieces (leaves of the tree) are searched in the text. Each time a leaf reports an occurrence, its parent node checks the area looking for its pattern (whose size is about twice the size of the leaf pattern). Only if the parent node finds the longer pattern, it reports the occurrence to its parent, and so on. The occurrences reported by the root of the tree are the final answers.

This construction is correct because the partitioning lemma applies to each level of the tree, i.e. any occurrence reported by the root node *must* include an occurrence reported by one of the two halves, so we search both halves. The argument applies then recursively to each half.

Figure 4.6 illustrates this concept. If we search the pattern "aaabbbcccddd" with four errors in the text "xxxbbxxxxxxx", and split the pattern in four pieces to be searched with one error, the piece "bbb" will be found in the text. In the original approach, we would verify the complete pattern in the text area, while with the new approach we verify only its parent "aaabbb" and immediately determine that there cannot be a complete match.

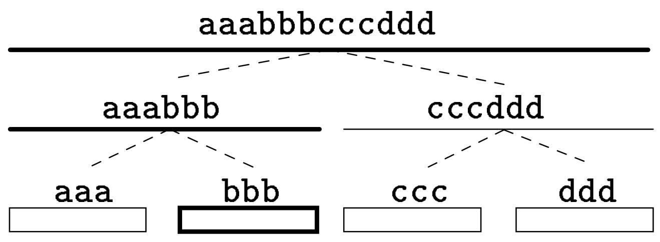


Figure 4.6: The hierarchical verification method for a pattern split in 4 parts. The boxes (leaves) are the elements which are really searched, and the root represents the whole pattern. At least one pattern at each level must match in any occurrence of the complete pattern. If the bold box is found, all the bold lines may be verified.

If $j$ is not a power of two we try to build the tree as well balanced as possible. This is because an unbalanced tree will force the verification of a long pattern because of the match of a short pattern (where the long pattern is more than twice as long as the short one). The same argument shows that it is not a good idea to use ternary or higher arity trees. Finally, we could increase $j$ to have a perfect binary partition, but shorter pieces trigger more verifications.

In order to handle partitions which are not a power of two, we need the strong version of the Partitioning Lemma of the previous section. For instance, if we determine $j = 5$, we have to partition the tree in, say, a left child with three pieces and a right child with two pieces. The standard partitioning lemma tells us that each subtree could search its pattern with $\lfloor k/2 \rfloor$ errors, but this will increase the verifications of the subtree with the shorter pattern. In fact, we can search the left subtree with $\lfloor 3k/5 \rfloor$ errors and the right one with $\lfloor 2k/5 \rfloor$ errors. Continuing with this technique we arrive to the leaves, which are searched with $\lfloor k/5 \rfloor$ errors as expected.

In subsequent chapters we apply this hierarchical verification method in different cases and show its practical improvements over the simple verification algorithm. We now analyze the cost of this verification algorithm.

We analyze the average amount of verification work to perform per character for each subpattern. We assume that $j = 2^s$ for integer $s$, and that $m = 2^s \ell$. Hence, we search $j$ subpatterns of length $\ell$. Recall that since we search patterns of length $m/j$ with $k/j$ errors, the error level $\alpha$ is essentially the same for any subproblem. Hence, the probability for any subpattern of length $m'$ to match in the text is $O(\gamma^{m'})$ (where $\gamma < 1$, Eq. (4.1)). To verify the occurrence of a pattern of length $\ell = m/j$, an area of length $\ell' = m/j + 2k/j = O(\ell)$ must be checked. The cost of that verification is $O(\ell^2)$ (at most, since we assume the use of plain dynamic programming[5]).

Consider a given subpattern of length $\ell$. It matches a given text position with probability $O(\gamma^\ell)$. This match causes its parent to perform a verification in an area of length $2\ell'$ (since the parent is of length $2\ell$). With some probability the parent pattern is not found and the verification ends here. Otherwise the parent is found and we must proceed upward in the tree. The probability of having to continue the verification is

$$Pr(\text{parent node / child node}) \;=\; \frac{Pr(\text{parent} \wedge \text{child})}{Pr(\text{child})} \;\leq\; \frac{Pr(\text{parent})}{Pr(\text{child})} \;=\; \frac{\gamma^{2\ell}}{\gamma^\ell} \;=\; \gamma^\ell$$

and therefore with probability at most $\gamma^\ell$ we pay the next verification which spans an area of length $4\ell'$, and so on. Notice that in the next verification we will find the longer pattern with probability $\gamma^{2\ell}$.

This process continues until we either find the complete pattern or we fail to find a subpattern. The total amount of work to perform is

$$\gamma^\ell \;(\;(2\ell)^2 \;+\; \gamma^\ell\;(\;(4\ell)^2 \;+\; \gamma^{2\ell}\;(\;(8\ell)^2 \;+ \ldots))) = \gamma^\ell\;(2\ell)^2 \;+\; \gamma^{2\ell}\;(4\ell)^2 \;+\; \gamma^{4\ell}\;(8\ell)^2 \;+ \ldots$$

which formally is

$$\sum_{i=1}^{s} \gamma^{\ell 2^{i-1}}\;(2^i \ell)^2 \;=\; 4\ell^2 \sum_{i=0}^{s-1} 4^i \left(\gamma^\ell\right)^{2^i}$$

The summation can be bounded with an integral to find that it is between $C(\ell) - C(m)$ and $\gamma^\ell + C(\ell) - C(m/2)$, where $C(x) = \gamma^x(x\ln(1/\gamma) - 1)/(\ell^2 \ln^2(1/\gamma)\ln 2)$. Therefore, as only the first term of the summation is $4\ell^2\gamma^\ell$, the summation is $\Theta(\gamma^\ell)$ and the total verification cost is

$$O(\ell^2\;\gamma^\ell) \;=\; O((m/j)^2\gamma^\ell) \tag{4.7}$$

which is much better than $O(m^2\gamma^\ell)$, which is the cost when the verification is not hierarchical.

We now generalize the above analysis by assuming that the verification tree is pruned when the patterns are of length $\ell r$, and the subtrees are replaced by arbitrary objects which match with probability $p$ (this is of interest later in this work).

[5]Since this area is already suspicious, we cannot ensure a random letter distribution that could encourage the use of better algorithms on average.

Once a leaf is found, we must verify an area of length $\ell r$ to determine whether their concatenation appears. That concatenation is found with probability $O(\gamma^{\ell r})$. When it is found, its parent is verified (an area of length $2\ell r$), which continues the verification with probability $\gamma^{2\ell r}$. Hence, the verification cost per piece is

$$p\ 4(\ell r)^2\ (1+\gamma^{\ell r}+C(\ell r)-C(m/2)) \ = \ O(p\ (\ell r)^2(1+\gamma^{\ell r})) \ = \ O(p\ (\ell r)^2) \qquad (4.8)$$

### 4.3.2 Superimposed Searching

Another kind of hierarchical verification is used in this thesis: given a number $r$ of patterns, we are able to superimpose them in such a way that we perform a single search. The probability of such pattern matching the text is the same as for a single pattern, except because the alphabet size is reduced from $\sigma$ to $\sigma/r$. Moreover, when we find a match, we are not able to say which is the pattern that actually matched (in fact it is possible that none matched). Hence, we must verify all the patterns in the area to determine which matched, if any.

An alternative to check all the $r$ patterns is to apply hierarchical verification. Assume that $r$ is a power of two. Then, when the "superimposed" search reports a match, run two new searches over the suspicious area: one which superimposes the first half of the patterns and another with the second half. Repeat the process recursively with each of the two searches that finds again a match. At the end, the searches will represent single patterns and if they find a match we know that that pattern has been really found (see Figure 4.7). Of course the preprocessing for the required subsets of patterns are done beforehand (they are $2r-1=O(r)$, so the space and preprocessing cost does not change). If $r$ is not a power of two then one of the halves may have one more pattern than the other.

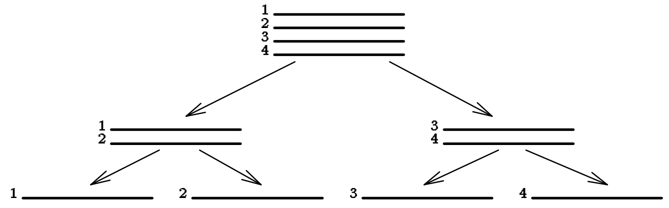


Figure 4.7: The hierarchical verification method for 4 patterns superimposed. Each node of the tree represents a check (the root which represents in fact the global filter). If a node passes the check, its two children are tested. If a leaf passes the check, its pattern has been found.

The advantage of this hierarchical verification is that it can remove a number of candidates from consideration in a single test. Moreover, it can even find that no pattern has really matched before actually checking any specific pattern (i.e. it may happen that none of the two halves match in a spurious match of the whole group). The worst-case overhead over plain verification is just a constant factor, that is, twice as many tests over the suspicious area ($2r-1$ instead of $r$).

On average, as we show now analytically and later experimentally (in Section 7.1, hierarchical verification is by far superior to plain verification.

Given an area of length $\ell$ to check for $r$ patterns, the naive verification will take $O(rT(\ell))$ time, where $T(\ell)$ is the verification cost for a single pattern. We analyze now the effect of this hierarchical verification. We call $\gamma_r^\ell$ the matching probability when $r$ patterns of length $\ell$ are superimposed.

Following Eq. (4.1), we have

$$\gamma_r \;=\; \left(\frac{e^2 r}{\sigma(1-\alpha)^2}\right)^{1-\alpha} \;=\; r^{1-\alpha}\gamma \tag{4.9}$$

where we notice that the old $\gamma$ corresponds now to $\gamma_1$.

For each match we have to verify an area of length $O(\ell)$ (at cost $T(\ell)$) for two sets of $r/2$ superimposed patterns each. Each set is found with probability $\gamma_{r/2}^{\ell}$ and so on. As before, we obtain the probability that a group of size $r/2$ matches given that the larger set matched is

$$P(\text{parent node} \;/\; \text{child node}) \;=\; \frac{P(\text{parent} \;\wedge\; \text{child})}{P(\text{child})} \;\leq\; \frac{P(\text{parent})}{P(\text{child})} \;=\; \frac{\gamma_{r/2}^{\ell}}{\gamma_r^{\ell}} \;=\; \frac{1}{2^{\ell(1-\alpha)}}$$

where we have used Eq. (4.9) for the last step. In particular $\gamma_{r/2^i} = \gamma_r/2^{i(1-\alpha)}$.

The total cost due to verifications is therefore

$$\gamma_r^{\ell}\left(2\;T(\ell)+2\;\frac{\gamma_{r/2}^{\ell}}{\gamma_r^{\ell}}\left(2\;T(\ell)+2\;\frac{\gamma_{r/4}^{\ell}}{\gamma_{r/2}^{\ell}}\ldots\right)\right) \;=\; 2T(\ell)\gamma_r^{\ell}+4T(\ell)\gamma_{r/2}^{\ell}+8T(\ell)\gamma_{r/4}^{\ell}+\ldots$$

or more formally

$$2T(\ell)\sum_{i=0}^{\log_2 r-1} 2^i\gamma_{r/2^i}^{\ell} \;=\; 2T(\ell)\gamma_r^{\ell}\sum_{i=0}^{\log_2 r-1} 2^{(1-\ell(1-\alpha))i}$$

and since the summation is $O(1)$ we have a total cost of $(\gamma_r^{\ell}T(\ell))$. Hence, we work $O(T(\ell))$ per verification instead of $O(rT(\ell))$ that we would work using a naive verification. This shows that we do not pay more per verification by superimposing more patterns (although the matching probability is increased).

Notice that we assumed that the summation is $O(1)$, which is not true for $\alpha \geq 1-1/\ell$. This happens only for very high error levels, which are of no interest in practice.

# Part I

# On-line Searching

This Part of the thesis is devoted to on-line approximate searching. This is the problem of finding a pattern in a text when there is no time to preprocess the text. Only the pattern can be preprocessed. This is the area which has received most attention from the community since the seventies and where it is most difficult to improve current results.

We present three chapters in this part. Chapter 5 is devoted to one of the best results of this thesis: a new bit-parallel algorithm which is currently the fastest for short patterns. Chapter 6 presents three different filtration algorithms, some of which are improvements over previously known versions. Our improvement over one of these is currently the fastest known algorithm for moderate error levels. At the end of Chapter 6 we study more in depth the technique of using a deterministic automaton which is partially built.

We present in Figure I.1 the "map" of the current best algorithms for each case of interest in text searching. The exact values correspond to English text, although similar figures are observed on random text for $\sigma = 8$ to 64. We have not studied in depth other cases, such as very long patterns or very small alphabets. As it can be seen, our new filtering algorithm of Section 6.1 is the best for moderate error ratios. For short patterns, our bit-parallel algorithm of Chapter 5 is the best. In the remaining area the best is Myers' new bit-parallel algorithm [Mye98], except for long patterns and high error ratios where the Four Russians technique of Wu, Manber and Myers [WMM96] remains the best. Hence, the algorithms of this thesis are the fastest ones for short patterns or for moderate error levels. Less interestingly, they are also the fastest for very high error levels.

It is important to notice that these results are "a posteriori". That is, the other algorithms we have created in this thesis have been the best in some areas at their times of publication (see the original papers). This is a very fast-moving and highly competitive area. For instance, a previous map of this kind can be found in [BYN98d]. One of the latest inventions in this area has been Myers' algorithm, which has swept under the carpet a number of previously outstanding algorithms.

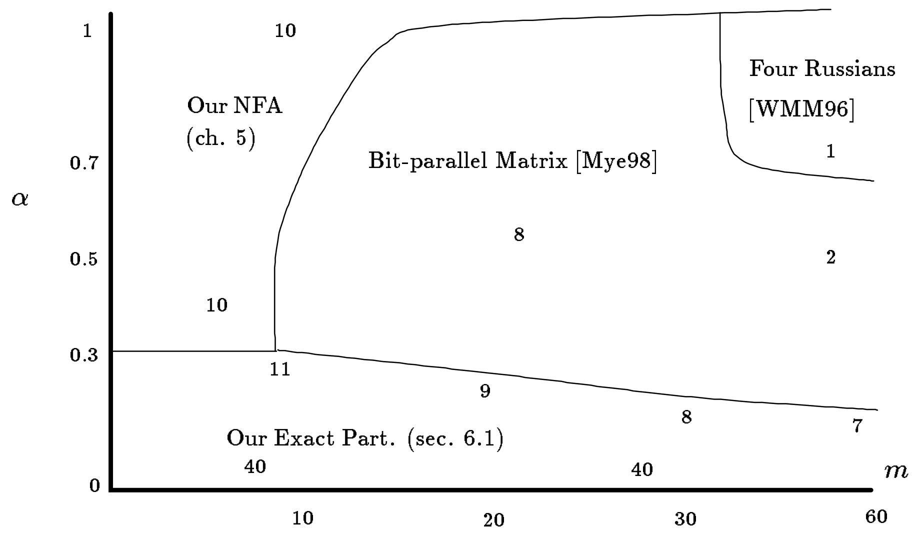
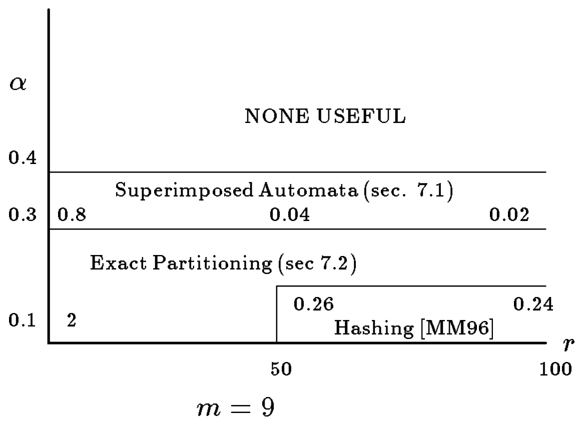
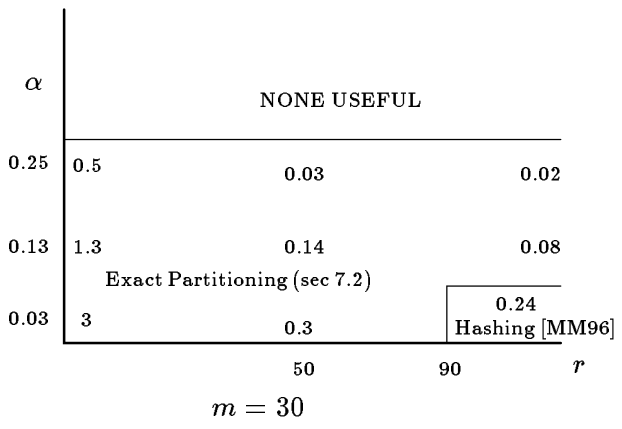


Figure I.1: The areas where each algorithm is the best for English text and $w = 32$. The numbers indicate megabytes per second in our machine.

Chapter 7, the third of this Part, deals with multiple patterns, that is, we have to search many patterns at the same time allowing errors. Despite having received less attention, this problem is very important and has applications to spelling, information retrieval and even to the classical problem of a single pattern, as we explain in the chapter. We present three new algorithms, which are carefully analyzed and experimentally tested. These are the only algorithms that exist for now for this problem when $k > 1$, and they also improve previous work (which exists only for $k = 1$) when the number of patterns is not larger than 50–150 (depending on the parameters of the problem). All the three algorithms are extensions of on-line single-pattern algorithms.

Figure I.2 gives a gross map of the best current algorithms for multipattern matching, for some choices of pattern length and number of patterns. The same observations made for Figure I.1 apply.





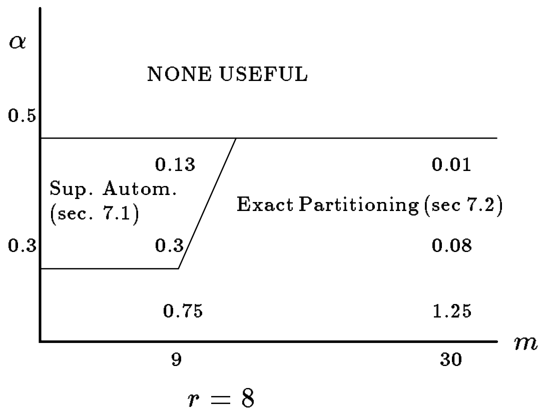


Figure I.2: The areas where each algorithm is the best for English text and $w = 32$. The numbers indicate megabytes per second in our machine.

# Chapter 5

# A Bit-Parallel Algorithm

In this chapter we present a new algorithm for on-line approximate string matching. The algorithm is based on the simulation using bit operations of a non-deterministic finite automaton built from the pattern and using the text as input. The running time achieved is $O(n)$ for small patterns (i.e. whenever $mk = O(w)$). Longer patterns can be processed by partitioning the automaton into many machine words, at $O(mk/w\ n)$ search cost. We allow generalizations in the pattern, such as classes of characters, gaps and others, at essentially the same search cost.

We then explore other novel techniques to cope with longer patterns. We show how to partition the pattern into short subpatterns which can be searched with less errors using the simple automaton, to obtain an average cost close to $O(\sqrt{mk/w}\ n)$. Moreover, we allow superimposing many subpatterns in a single automaton, obtaining near $O(\sqrt{mk/(\sigma w)}\ n)$ average complexity.

Finally, we combine analytical and experimental results to design an optimal heuristic to combine all the techniques. The experiments show that our algorithm is among the fastest for typical text searching, being the fastest in an important range of parameters of the problem.

This work has been published in [BYN96b, BYN96a, BYN98d, NBY98b].

## 5.1 A New Parallelization Technique

Consider again the NFA of Figure 2.5, which recognizes the approximate occurrences of the pattern in the text. This NFA has $(m+1) \times (k+1)$ states. We assign number $(i,j)$ to the state at row $i$ and column $j$, where $i \in 0..k, j \in 0..m$. Initially, the active states at row $i$ are at the columns from 0 to $i$, to represent the deletion of the first $i$ characters of the pattern.

Consider the boolean matrix $A$ corresponding to this automaton. $A_{i,j}$ is 1 if state $(i,j)$ is active and 0 otherwise. The matrix changes as each character of the text is read. The new values $A'_{i,j}$ can be computed from the current ones by the following rule

$$A'_{i,j} \;=\; (A_{i,j-1}\ \ \&\ \ (\ T_{current} = P_j\ ))\ |\ A_{i-1,j}\ |\ A_{i-1,j-1}\ |\ A'_{i-1,j-1} \qquad (5.1)$$

which is used for $i \in 0..k, j \in 1..m$. If $i = 0$ only the first term is used. Note that the empty transitions are represented by immediately propagating a 1 at any position to all the elements following it in its diagonal, all in a single iteration (thus computing the $\varepsilon$-closure). The self-loop at the initial state is represented by the fact that column $j = 0$ is never updated.

It is not hard to prove by induction that once an automaton state is active, then all the states below it in the same column are active too. The same happens to the states below it in the same diagonal, and in fact with all the states in the triangle limited by the column and the diagonal.

It is then possible to describe the automaton by the minimum rows of the states active at each column. This observation allows relating the automaton to the classical dynamic programming matrix (see Section 2.2). Notice that what the current column of the dynamic programming matrix stores at position $j$ is the minimum row of this NFA which is active in column $j$ of the automaton.

In this sense, the dynamic programming matrix corresponds to *simulating* this automaton by columns (i.e. packing columns in machine words) [BY96]. On the other hand, the work of Wu and Manber ([WM92a], see Section 3.1.4) consists fundamentally in simulating this automaton by rows (packing each row in a machine word). In both cases, the dependencies introduced by the diagonal empty transitions prevent the parallel computation of the new values. In [BY91] it was shown that this dependence can be avoided by simulating the automaton by by *diagonals*, such that each diagonal captures the $\varepsilon$-closure. However, the associated bit-parallel update formulas are more complex and no solution was devised in [BY91]. We present next a realization of this idea, by finding a constant-time update formula. This achievement leads to a new and fast algorithm.

Suppose we use just the *full* diagonals of the automaton (i.e. those of length $k+1$). See Figure 5.1. This presents no problem, since those (shorter) diagonals below the full ones always have value 1, while those past the full ones do not influence state $(m, k)$. The last statement may not be obvious, since the vertical transitions allow carrying 1's from the last diagonals to state $(m, k)$. However, each 1 present at the last diagonals must have crossed the last full diagonal, where the empty transitions (deletions) would have immediately copied it to the state $(m, k)$. That is, any 1 that goes again to state $(m, k)$ corresponds to a segment containing one that has already been reported [1].

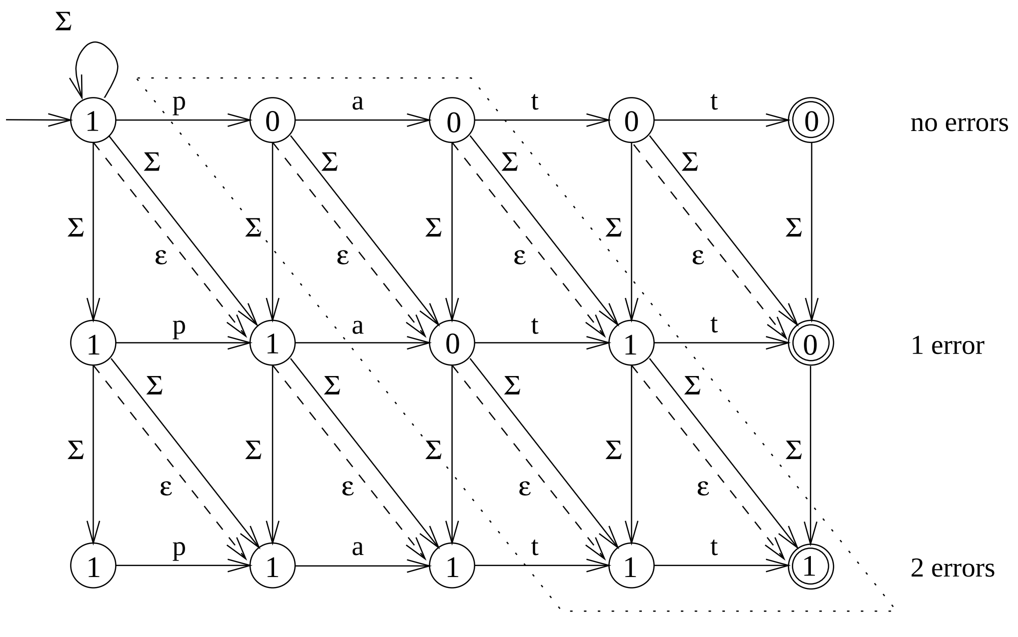


Figure 5.1: Our representation of the NFA. We enclose in dotted lines the states actually represented in our algorithm.

[1]If reporting those spurious matches is required we just have to check the right neighborhood of the positions reported with this algorithm using a wider automaton that includes the last diagonals. This also allows telling the exact number of errors of each match. As we show later, this horizontal extension, even if it requires extra machine words, is not significatively more expensive.

As already mentioned, whenever state $(i,j)$ is active, states $(i+d,j+d)$ are also active for all $d>0$ (due to the $\varepsilon$ transitions). Thus, if we number diagonals regarding the column they begin at, the state of each diagonal $i$ can be represented by a number $D_i$, the smallest row value active in that diagonal (i.e. the smallest error). Then, the state of this simulation consists of $m-k+1$ values in the range $0..k+1$. Note that $D_0$ is always 0, hence there is no need to store it.

The new values for $D_i$ ($i \in 1..m-k$) after we read a new text character $c$ are derived from Eq. (5.1)

$$D'_i \;=\; \min(\; D_i+1, \;\; D_{i+1}+1, \;\; g(D_{i-1},c) \;) \tag{5.2}$$

where $g(D_i,c)$ is defined as

$$g(D_i,c) \;=\; \min(\; \{k+1\} \;\cup\; \{\; j \;/\; j \geq D_i \;\wedge\; P_{i+j}=c \;\} \;)$$

The first term of the $D'$ update formula represents a substitution, which follows the same diagonal. The second term represents the insertion of a character (coming from the next diagonal above). Finally, the last term represents matching a character: we select the minimum active state (hence the min of the $g$ formula) of the previous diagonal that matches the text and thus can move to the current one. The deletion transitions are represented precisely by the fact that once a state in a diagonal is active, we consider that all subsequent states on that diagonal are active (so we keep just the minimum). The empty initial transition corresponds to $D_0=0$. Finally, we find a match in the text whenever $D_{m-k} \leq k$.

This simulation has the advantage that it can be computed in parallel for all $i$, since there is no dependency between the current values of different diagonals. We use this property to design a fast algorithm that exploits bit-parallelism for small patterns, and then extend it to handle the general case.

## 5.2 A Linear Algorithm for Small Patterns

We show in this section how to simulate the automaton by diagonals using bit-parallelism, assuming that our problem fits in a single machine word. We first select a suitable representation for our problem and then describe the algorithm.

Since we have $m-k$ non-trivial diagonals, and each one takes values in the range $0..k+1$, we need at least $(m-k)\lceil \log_2(k+2)\rceil$ bits. However, we cannot efficiently compute the $g$ function in parallel for all $i$ with this optimal representation. We could precompute and store it, but it would take $O(\sigma(k+1)^{m-k})$ space if it had to be accessed in parallel for all $i$. At this exponential space cost, the automaton approach of Section 3.1.2 is preferable.

Therefore, we use a unary encoding for the $D_i$ values, since in this case $g$ can be computed in parallel. Thus, we need $(m-k)(k+2)$ bits to encode the problem, where each of the $m-k$ blocks of $k+2$ bits stores the value of a $D_i$. Then we must have $(m-k)(k+2) \leq w$ (where we recall that $w$ is the length in bits of the computer word). We address later the case where this does not hold.

Each value of $D_i$ is stored as 1's aligned to the right of its $(k+2)$-wide block (thus there is a separator at the highest bit always having 0). The blocks are stored contiguously, the last one ($i=m-k$) aligned to the right of the computer word. Thus, our bit representation of state $D_1, ..., D_{m-k}$ is

$$D \;=\; 0\; 0^{k+1-D_1}\; 1^{D_1} \;\; 0\; 0^{k+1-D_2}\; 1^{D_2} \;\; ... \;\; 0\; 0^{k+1-D_{m-k}}\; 1^{D_{m-k}}$$

where we use exponentiation to denote digit repetition.

Until now we have used a *numerical* view of the process, where the state of the search is represented by a sequence of numbers $D_i$ and some arithmetical operations are performed on them. There is, on the other hand, a more *logical* view, where we just consider the automaton as having active (1) and inactive (0) states, so that the 1's move across the arrows as they can.

In fact, what our word contains is a rearrangement of the 0's and 1's of (the relevant part of) the automaton. It is important to notice that the rearrangement *exchanges* 0's and 1's and reads the diagonals left-to-right and upwards (see $D$ in Figure 5.2).

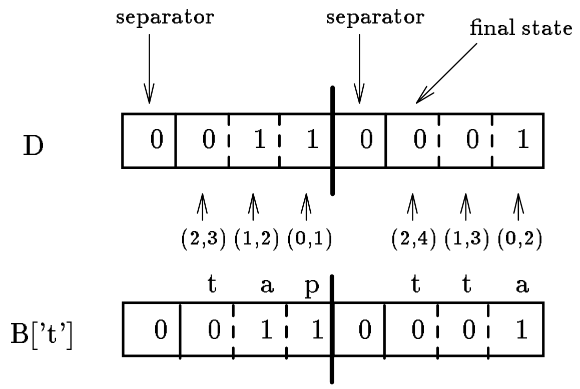


Figure 5.2: Encoding of the example NFA. In this example, $b['\mathtt{t}'] = 0011$.

As both forms of reading the formulas are equivalent and direct, we encourage the reader to consider the formulas that follow under these two alternative interpretations, as one may be easier to grasp than the other depending on the reader. We make more explicit now how are these two interpretations:

**Numerical:** consider our bit arrangement as a sequence of *numbers* $D_i$ represented in unary notation. Interpret the operations as arithmetical manipulations.

**Logical:** consider our bit mask as a mapping from the automaton states to bit positions (recall that 0's and 1's are interchanged). Interpret the operations as bits that move from one place to another.

With our bit representation, taking minimum is equivalent to *and*ing, adding 1 is equivalent to shifting one position to the left and *or*ing with a 1 at the rightmost position, and accessing the next or previous diagonal means shifting a block ($k+2$ positions) to the left or right, respectively.

The computation of the $g$ function is carried out by defining, for each character $c$, an $m$ bits long mask $b[c]$, representing match (0) or mismatch (1) against the pattern, and then computing a mask $B[c]$ having at each block the $(k+1)$ bits long segment of $b[c]$ that is relevant to that block (see Figure 5.2). That is,

$$b[c] \;=\; (c \neq P_m)\;(c \neq P_{m-1})\;\ldots\;(c \neq P_1) \qquad (5.3)$$

where each condition stands for a bit and they are aligned to the right. So we precompute

$$B[c] \;=\; 0\; s_{k+1}(b[c],0)\;\; 0\; s_{k+1}(b[c],1)\;\; \ldots\;\; 0\; s_{k+1}(b[c],m-k-1)$$

for each $c$, where $s_j(x,i)$ shifts $x$ to the right in $i$ bits and takes the last $j$ bits of the result (the bits that "fall" are discarded). Note that $B[c]$ fits in a computer word if the problem does.

We have now all the elements to implement the algorithm. We represent the current state by a computer word $D$. The value of all $D_i$'s is initially $k+1$, so the initial value of $D$ is $D_{in} = (0\ 1^{k+1})^{m-k}$. The formula to update $D$ upon reading a text character $c$ is derived from Eq. (5.2)

$$\begin{aligned} D' \quad = & \quad (D << 1) \mid (0^{k+1}1)^{m-k} \\ & \& \ (D << (k+3)) \mid (0^{k+1}1)^{m-k-1}0\ 1^{k+1} \\ & \& \ (((x + (0^{k+1}1)^{m-k}) \ \wedge \ x) >> 1) \\ & \& \ D_{in} \qquad\qquad\qquad\qquad\qquad\qquad (5.4) \\ \text{where} \qquad x \quad = & \quad (D >> (k+2)) \mid B[c] \end{aligned}$$

We explain now how this formula is obtained:

- The update formula is a sequence of *and*'s, corresponding to the *min* of Eq. (5.2). For instance, if we represent 3 and 5 in unary using 8 bits, we have 00000111 and 00011111, respectively. If we perform an *and* on these representations, we get 00000111 & 00011111 = 00000111, which is the representation of 3, i.e. $\min(3,5)$.

- The first line corresponds to $D_i + 1$. This is obtained by shifting the bits to the left one position and putting a 1 in the "hole" left by the shift. For instance, if we do this to the representation of 4 in eight bits (00001111) we get (00001111 << 1) | 00000001 = 00011111, which is the representation of 5. This operation occurs inside each one of the $m-k$ diagonals represented.

- The second line corresponds to $D_{i+1} + 1$. In this case the shift is by $k+3$ positions, from which $k+2$ account for bringing the data of the next diagonal to the position of the current one, and the extra position shifted does the effect of the $+1$ as in the previous item. In this case we also put the 1 to fill all the holes, except for the last diagonal which does not get anything from the next one. The formula sets this last diagonal in $01^{k+1}$, which corresponds to all inactive states.

- The third line is the $g$ function applied to the previous diagonal. First, $x$ is obtained by taking the states of the previous diagonal (hence the shift to the right in $k+2$ positions) and *or*-ing them with the mask $B[c]$, where $c$ is the text character read and $B$ aligns with the mapping from automaton states to bit positions. The effect of the *or* is that only the bits which were already active (i.e. zero) and can cross the horizontal arrow of the automaton (i.e. have $B$ in zero in the proper place) remain active (zero). The rest of the processing on $x$ propagates the active states down to the rest of the states of the diagonal (i.e. $\varepsilon$-transitions). In our representation this means that we want the zero of the lowest bit position to be propagated to all the higher bit positions. This is obtained by adding 1, which will invert all the bits, from lowest to highest, until a zero is found. That zero is also inverted but the inversion is not propagated anymore. Finally, the "$\wedge$" (exclusive or) operation compares two numbers bitwise and sets to 0 the positions where the bits are equal and to 1 the others. If we compare the original $x$ and the new result of adding 1, we have that all the bits from the lowest one to the first zero (inclusive) differ, and the rest are equal. Since we wanted zeros from the first zero of $x$ to the highest bit, it is just a matter of shifting one position to the right. For instance, if $x$ = 01001011, we add 1 to obtain 01001011 + 00000001 = 01001100, and then the exclusive-or yields 01001100 $\wedge$ 01001011 = 00000111, which after shifting one position to the right becomes 00000011. This last number represents the propagation of the lowest zero of $x$ to the left.

- Finally, the fourth line ensures the invariant of having zeros in the separators (needed to limit the propagation of "+").

Note that we are assuming that the shifts get zeros from both borders of the word (i.e. unsigned semantics). If this is not the case in a particular architecture, additional masking is necessary [2].

We detect that state $(m, k)$ is active by checking whether $D$ & $(1 << k)$ is 0. When we find a match, we clear the last diagonal. This ensures that our occurrences always end with a match.

### 5.2.1 A Simple Filter

We can improve the previous algorithm (and in fact most other algorithms as well) by noticing that any approximate occurrence of the pattern with $k$ errors must begin with one of its $k+1$ first characters. This allows quickly discarding large parts of the text performing very few operations per character.

We do not run the automaton through all text characters, but scan the text looking for any of the $k+1$ initial characters of the pattern. Only then do we start the automaton. When the automaton returns to its initial configuration, we resume the scanning. The scanning is much cheaper than the operation of our automaton, and in fact it is cheaper than the work done per text character in most other algorithms (except possibly filtration algorithms).

We precompute a boolean table $S[c]$, that stores for each character $c$ whether it is one of the first $k+1$ letters of the pattern. Observe that this table alone solves the problem for the case $k = m-1$ (since each positive answer of $S$ is an occurrence).

### 5.2.2 The Code

Figure 5.3 presents the complete algorithm (i.e. using the automaton plus the filter). For simplicity, we do not refine the preprocessing, which can be done more efficiently than the code suggests.

## 5.3 Handling Extended Patterns

We show now that some of the extended patterns considered in [BYG92, WM92a, WMM96] and explained Section 3.2.1 in can be introduced into our algorithm at no additional search cost.

As in the Shift-Or algorithm for exact matching [BYG92], we can specify classes of characters. This is achieved by modifying the $b[]$ table (5.3), making any element of the set to match that position, with no running time overhead. That is, if $P = P_1 ... P_m$ where now $P_i \subseteq \Sigma$, then we replace the condition $(c \neq P_i)$ of (5.3) by $(c \notin P_i)$.

In addition to classes of characters, we can support the # operator as defined in [WM92a]. That is, $x\#y$ allows zero or more arbitrary characters among the strings $x$ and $y$ in the occurrences. Those characters are not counted as errors. As shown in [WM92a], in order to handle this operator we must enforce that whenever an automaton state in column $|x|$ is active, it keeps active from then on.

[2]Although programming language standards such as ANSI C ensure this property, those specifications are not always followed by all implementations, especially when the underlying architecture does not favor it.

```
search $(T,n,P,m,k)$
 {        /* preprocessing */
   for each $c \in \Sigma$
     { $b[c] \;=\; (c \neq P_m)\ (c \neq P_{m-1})\ \ldots\ (c \neq P_1)$
       $B[c] \;=\; 0\ s_{k+1}(b[c],0)\ \ 0\ s_{k+1}(b[c],1)\ \ldots\ \ 0\ s_{k+1}(b[c],m-k-1)$
       $S[c] \;=\; (c \in P_{1..k+1})$
     }
   $D_{in} = (0\ 1^{k+1})^{m-k}$
   $M1 = (0^{k+1}1)^{m-k}$
   $M2 = (0^{k+1}1)^{m-k-1}\ 0\ 1^{k+1}$
   $M3 = 0^{(m-k-1)(k+2)}\ 0\ 1^{k+1}$
   $G = 1 << k$
          /* searching */
   $D = D_{in}$
   $i = 0$
   while $(++\,i <= n)$
      if $(S[T_i])$          /* is one of the first $k+1$ characters?  */
         do { $x = (D >> (k+2))\ |\ B[T_i]$
              $D = ((D << 1)\ |\ M1)\ \&\ ((D << (k+3))\ |\ M2)$
                   $\&\ (((x + M1)\ \wedge\ x) >> 1)\ \&\ D_{in}$
              if $(D\ \&\ G == 0)$
                 { report a match ending at $i$
                   $D = D\ |\ M3$          /* clear last diagonal */
                 }
            }
         while $(D\ != D_{in}\ \&\&\ ++i <= n)$
 }
```

Figure 5.3: Algorithm to search for a short pattern. Recall that strings start at position 1.

Hence, to search for $x_1\#x_2\#...\#x_t$ we create a $D_{\#}$ word having all 1's except at all states of columns $|x_1|$, $|x_1|+|x_2|$, ..., $|x_1|+|x_2|+...+|x_{t-1}|$. We now modify the computation of $x$ in Eq. (5.2), which becomes

$$x \;=\; ((D >> (k+2))\ |\ B[c])\ \&\ (D\ |\ D_{\#})$$

(clearly this technique is orthogonal to the use of classes of characters).

We can modify the automaton to compute the edit distance (more precisely, determine whether the edit distance is $\leq k$ or not). This is obtained by eliminating the initial self-loop and initializing the automaton at $D = D_{in}$. However, we need to represent the $k+1$ initial diagonals that we discarded. If we need the exact edit distance, we must also represent the last $k$ diagonals that we discarded. If there is no a priori bound on the distance, we need to set $k = m$.

We can search for whole words, running the edit-distance algorithm only from word beginnings (where we re-initialize $D = D_{in}$), and checking matches only at the end of words.

Searching with different integral costs for insertion and substitution (including not allowing such operations) can be accommodated in our scheme, by changing the arrows. Deletion is built into the model in such a way that in order to accommodate it we must change the meaning of our "diagonals", so that they are straight $\varepsilon$-transition chains.

Other generalizations are studied in [WM92a]. We can handle them too, although not as easily and efficiently as the previous ones. One such generalization is the combination in the pattern of parts that must match exactly with others that can match with errors. The adaptation to avoid propagation of $\varepsilon$-closures in our scheme is ad-hoc and not as elegant as in [WM92a]. However, we believe that the most effective way to handle these patterns is to quickly search for the parts that match exactly and then try to extend those matches to the complete pattern, using the automaton to compute edit distance.

Another such generalization is the approximate search of regular expressions. In [WM92a], the regularities among rows allow solving any regular expression of $m$ letters using $\lceil m/8 \rceil$ or even $\lceil m/16 \rceil$ operations per text character, using $\lceil m/8 \rceil 2^8 \lceil m/w \rceil$ or $\lceil m/16 \rceil 2^{16} \lceil m/w \rceil$ machine words of memory, respectively. Our automaton is not so regular, and we would need roughly $O(k^2)$ times the space requirements and operations per text character of [WM92a]. To be more precise, in our scheme their formulas are still valid provided we replace $m$ by $(m-k)(k+2)$. For instance, at the cost they pay for $m \leq 32$, we can only solve for $m \leq 9$. However, our scheme is still reasonably applicable for short expressions.

## 5.4 Partitioning Large Automata

If the automaton does not fit in a single word, we can partition it using a number of machine words for the simulation.

First suppose that $k$ is small and $m$ is large. Then, the automaton can be "horizontally" split into as many subautomata as necessary, each one holding a number of diagonals. We call "d-columns" those sets of diagonals packed in a single machine word. Those subautomata behave differently than the simple one, since they must communicate their first and last diagonals with their neighbors. Thus, if $(m-k)(k+2) > w$, we partition the automaton horizontally in $J$ d-columns, where $J = \lceil (m-k)(k+2)/w \rceil$. Note that we need that at least one automaton diagonal fits in a single machine word, i.e. $k+2 \leq w$.

Suppose now that $k$ is large (close to $m$, so that the width $m-k$ is small). In this case, the automaton is not wide but tall, and a vertical partitioning becomes necessary. The subautomata behave differently than the previous ones, since we must propagate the $\varepsilon$-transitions down to all subsequent subautomata. In this case, if $(m-k)(k+2) > w$, we partition the automaton vertically in $I$ d-rows (each d-row holding some automaton rows of all diagonals), where $I$ has the same formula as $J$. The difference is that, in this case, we need that at least one automaton row fits in a machine word, i.e. $2(m-k) \leq w$ (the 2 is because we need an overflow bit for each diagonal of each cell).

When none of the two previous conditions hold, we need a generalized partition in d-rows and d-columns. We use $I$ d-rows and $J$ d-columns, so that each cell contains $\ell_r$ bits of each one of $\ell_c$ diagonals. It must hold that $(\ell_r + 1)\ell_c \leq w$ (see Figure 5.4).

Simulating the automaton is now more complex, but it follows the same principle of the update formula (5.4). We have a matrix of automata $D_{i,j}$ ($i \in 0..I-1, j \in 0..J-1$), and a matrix of masks $B_{i,j}$ coming from splitting the original $B$. The new update formula for a text character $c$ is

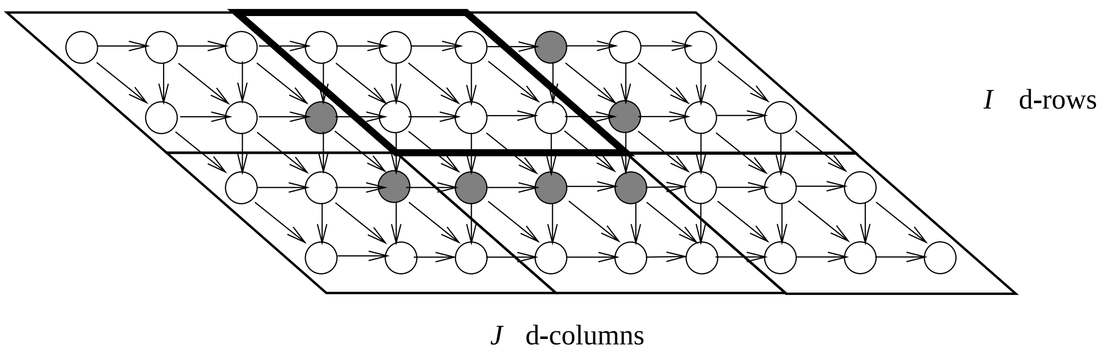


Figure 5.4: A $2 \times 3$ partitioned automaton where $\ell_c = 3$, $\ell_r = 2$, $I = 2$, $J = 3$. We selected a cell (bold edges) and shaded all the nodes of other cells affected by it. The bold-edged cell must communicate with those neighbors that own shaded nodes.

$$\begin{array}{rcl}
D'_{i,j} & = & (D_{i,j} << 1) \mid ((D_{i-1,j} >> (\ell_r - 1)) \ \& \ (0^{\ell_r}1)^{\ell_c}) \\
& & \& \ ((D_{i,j} << (\ell_r + 2)) \mid \\
& & \quad ((D_{i-1,j} << 2) \ \& \ (0^{\ell_r}1)^{\ell_c}) \mid \\
& & \quad (D_{i-1,j+1} >> ((\ell_r + 1)(\ell_c - 1) + \ell_r - 1)) \mid \\
& & \quad (D_{i,j+1} >> ((\ell_r + 1)(\ell_c - 1) - 1))) \\
& & \& \ (((x + (0^{\ell_r}1)^{\ell_c}) \ \wedge \ x) >> 1) \\
& & \& \ D_{in} \\
\text{where} \quad x & = & ((D_{i,j} >> (\ell_r + 1)) \mid (D_{i,j-1} << (\ell_r + 1)(\ell_c - 1)) \mid B_{i,j}[c]) \\
& & \& \ ((D'_{i-1,j} >> (\ell_r - 1)) \mid (1^{\ell_r}0)^{\ell_c})
\end{array}$$

and it is assumed $D_{-1,j} = D_{i,J} = 1^{(\ell_r+1)\ell_c}$ and $D_{i,-1} = 0^{(\ell_r+1)\ell_c}$. We find a match whenever $D_{I-1,J-1}$ has a 0 in its last position, i.e. at $(k - \ell_r(I-1)) + (\ell_r + 1)(\ell_c J - (m-k))$, counting from the right. In that case, we must clear the last diagonal, i.e. that of $D_{i,J-1}$ for all $i$.

If we divide the automaton in $IJ$ subautomata ($I$ d-rows and $J$ d-columns), we must update $I$ cells at each d-column. However, we use the cut-off heuristic of [Ukk85b] (see Section 3.1.1), i.e. we do not process the $m$ columns but only up to the last active one. That is, we work only on active automaton diagonals. Figure 5.5 illustrates.

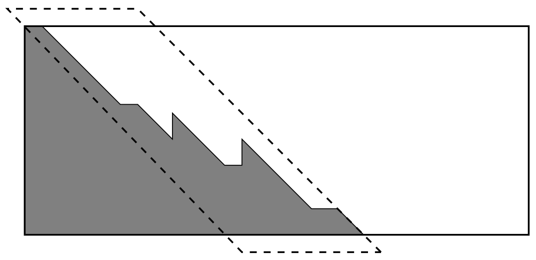


Figure 5.5: Illustration of active diagonals. The shaded areas are the active states of the automaton. It is only necessary to work on the area surrounded by dashed lines.

There are many options to pick $(I, J)$ for a given problem. The best choice is a matter of optimization which we address later. All the techniques to handle extended patterns are easily implemented in a partitioned automaton.

## 5.5 Partitioning the Pattern

The Partitioning Lemma proved in Section 4.2 allows us to reduce the number of errors if we divide the pattern, provided we search all the subpatterns. Each match of a subpattern must be checked to determine if it is in fact a complete match (notice that the subpatterns can be extended patterns themselves). Suppose we find at position $T_{..i}$ the end of a match for the subpattern ending at $P_{..j}$. Then, the potential match must be searched in the area $T_{i-j+1-k..i-j+1+m+k}$, an $(m+2k)$-wide area. This checking must be done with an algorithm resistant to high error levels, such as our automaton partitioning technique. As we show later, for moderately long patterns this is faster than plain dynamic programming and the cut-off heuristic of [Ukk85b]. Recall that we use the hierarchical verification technique explained in Section 4.3.1.

To perform the partition, we pick an integer $j$, and split the pattern into $j$ subpatterns of length $m/j$ (more precisely, if $m = qj + r$, with $0 \leq r < j$, $r$ subpatterns of length $\lceil m/j \rceil$ and $j - r$ of length $\lfloor m/j \rfloor$). Because of the Lemma, it is enough to check if any of the subpatterns is present in the text with at most $\lfloor k/j \rfloor$ errors.

If we partition the pattern in $j$ parts, we have to perform $j$ searches. Moreover, those searches will together trigger more verifications as $j$ grows (i.e. a piece split in two will trigger all the verifications triggered by the original piece plus spurious ones). This fact is reflected in the formula for the match probability of Section 4.1 (Eq. (4.1)), since the match probability is now $O(\gamma^{m/j})$, which may be much larger than $O(\gamma^m)$ even for a single piece. Therefore, we prefer to keep $j$ small.

A first alternative is to make $j$ just large enough for the subproblems to fit in a computer word, that is

$$j^* = \min \left\{ j \;/\; \left( \left\lceil \frac{m}{j} \right\rceil - \left\lfloor \frac{k}{j} \right\rfloor \right) \left( \left\lfloor \frac{k}{j} \right\rfloor + 2 \right) \leq w \;\;\wedge\;\; \left\lfloor \frac{m}{j} \right\rfloor > \left\lfloor \frac{k}{j} \right\rfloor \right\} \qquad (5.5)$$

where the second condition avoids searching a subpattern of length $m'$ with $k' = m'$ errors (those of length $\lceil m/j \rceil$ are guaranteed to be longer than $\lfloor k/j \rfloor$ if $m > k$). Such a $j^*$ always exists if $k < m$.

A second alternative is to use a smaller $j$ (and therefore the automata still do not fit in a computer word) and combine this technique with automaton partitioning for the subpatterns.

Figure 5.6 shows the general algorithm, which is written in that way for clarity. In a practical implementation, it is better to run all subsearches in synchronization, picking at any moment the candidate whose initial checking position is the leftmost in the set, checking its area and advancing that subsearch to its next candidate position. This allows us to avoid re-verifying the same text because of different overlapping candidate areas, which is done by remembering the last checked position and keeping the state of the checking algorithm.

The effectiveness of this method is limited by the error level. If the subpatterns appear very often, we spend a lot of time verifying candidate text positions. In Section 5.7 we find out which is the maximal error level for this scheme to be efficient.

A special case of this algorithm occurs if we use $j = k+1$, since then the pieces are searched with zero errors. This is a qualitatively different algorithm which was previously known (see Section 3.1.3). We can perform a single multipattern search, using an Aho-Corasick machine [AC75] to guarantee $O(n)$ total search time, or an extension of the Boyer-Moore family which is faster for few patterns.

```
PatternPartition $(T, n, P, m, k)$
 { $j$ = min { $r$ / $(\lceil m/r \rceil - \lfloor k/r \rfloor)(\lfloor k/r \rfloor + 2) \leq w \ \wedge \ \lfloor m/r \rfloor > \lfloor k/r \rfloor$}
   if $(j = 1)$ search $(T, n, P, m, k)$
   else { $a = 1$
          for $r \in 0..j-1$
             { $len = (r < m \ \% \ j)$ ? $\lceil m/j \rceil$ : $\lfloor m/j \rfloor$
               $b = a + len - 1$
               for each position $i$ reported by search$(T, n, P_{a..b}, len, \lfloor k/j \rfloor)$
                  check the area $T_{i-b+1-k..i-b+1+m+k}$
               $a = b + 1$
 }      }     }
```

Figure 5.6: Algorithm for pattern partitioning.

Wu and Manber [WM92a] propose to use the multipattern version of the Shift-Or, whose advantage is flexibility in the queries. In particular, if the pattern is extended, the multipattern Shift-Or algorithm is the correct choice. Baeza-Yates and Perleberg, on the other hand, propose to extend a Boyer-Moore type algorithm [BYP96].

For non extended patterns, we preferred to use the Sunday algorithm [Sun90] extended to multipattern search. However, in Section 6.1 we elaborate more on this algorithm and find new improvements on it. Therefore, in this chapter we do not consider the partition into $k+1$ pieces as part of the bit-parallel algorithm.

## 5.6 Superimposing the Subpatterns

When the search is divided into a number of subsearches for smaller patterns $P^1, ..., P^r$, it is possible to avoid searching for each one separately. We describe a technique, called *superimposition*, to collapse a number of searches into a single one.

In our scheme, all patterns have almost the same length. If they differ (at most by one), we truncate them to the shortest length. Hence, all the automata have the same structure, differing only in the labels of the horizontal arrows.

The superimposition is defined as follows: we build the $b[]$ table for each pattern (Eq. (5.3)), and then take the bitwise-*and* of all the tables. The resulting $b[]$ table matches in the position $i$ with the $i$-th character of *any* pattern. We then build the automaton as before using this table. The resulting automaton accepts a text position if it ends an occurrence of a much more relaxed pattern (in fact it is an extended pattern), namely

$$\{P^1_1, ..., P^r_1\} \ ... \ \{P^1_m, ..., P^r_m\}$$

for example, if the search is for "patt" and "wait", the string "watt" is accepted with *zero* errors (see Figure 5.7). Each occurrence reported by the automaton has to be verified for all the patterns involved.

For a moderate number of patterns, this still constitutes a good filtering mechanism, at the same cost of a single search. Clearly, the relaxed pattern triggers many more verifications than the simple ones. This severely limits the amount of possible superimposition. However, as we show later, in

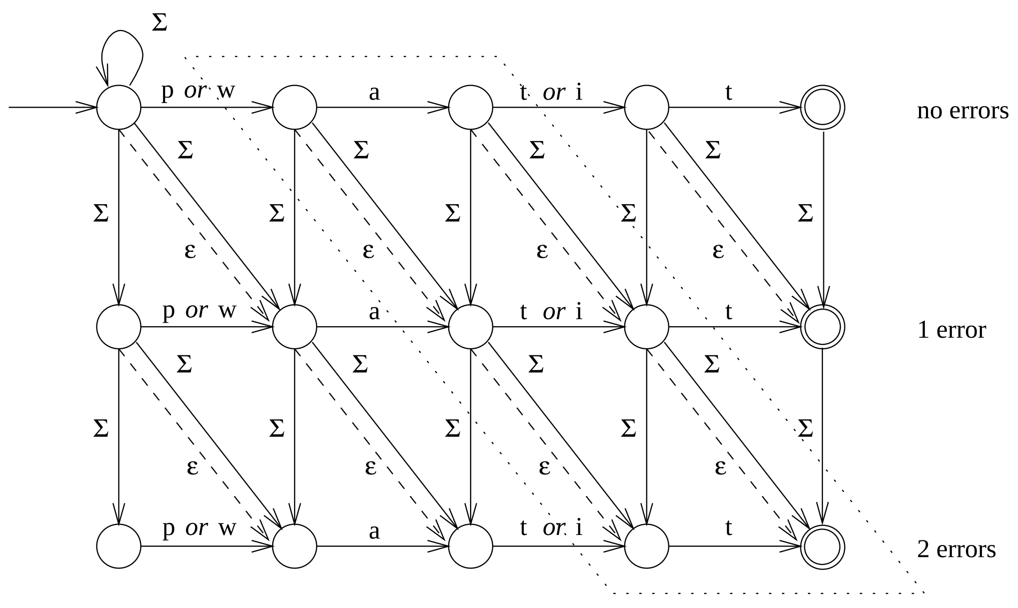


Figure 5.7: An NFA to filter the parallel search of "patt" and "wait".

practice this can cut the search times by a factor of two or three. This idea has been applied to multiple approximate string matching, where similar speedup figures were obtained (see Chapter 7).

We analyze later how many subpatterns can be collapsed while keeping the number of verifications low. We must then form sets of patterns that can be searched together, and search each set separately. If we use pattern partitioning in $j$ pieces and superimpose in groups of $r$ pieces, we must perform $\lceil j/r \rceil$ superimposed searches. We keep the groups of almost the same size, namely $r' = \lfloor j/\lceil j/r \rceil \rfloor$ and $r' = \lceil j/\lceil j/r \rceil \rceil$. Notice that extended patterns containing #'s may not be superimposed because their $D_{\#}$ words are different.

We group subpatterns which are contiguous in the pattern. When an occurrence is reported we cannot know which of the superimposed subpatterns caused the match (since the mechanism does not allow knowing), so we check whether the concatenation of the subpatterns appears in the area. From that point on, we use the normal hierarchical verification mechanism.

Observe that having the same character at the same position for two patterns improves the filter efficiency. This fact can be used to select the best partition of the pattern.

## 5.7 Analysis and Optimization

In this section we analyze the different aspects of our algorithms, and use this analysis to find the optimal choice for the many alternatives that have been left open. The theoretical analysis is intermingled with practical tuning sections and experiments showing their performance and confirming the analysis.

It is important to notice that our average-case analysis assumes that the pattern is not extended and that text and patterns are random strings over an independently generated and uniformly distributed alphabet of size $\sigma$. If the alphabet is not uniformly distributed we must replace the $\sigma$ in the formulas by $1/p$, where $p$ is the probability that two random letters match. For generalized patterns, the average values are different, but we are not able to compute them.

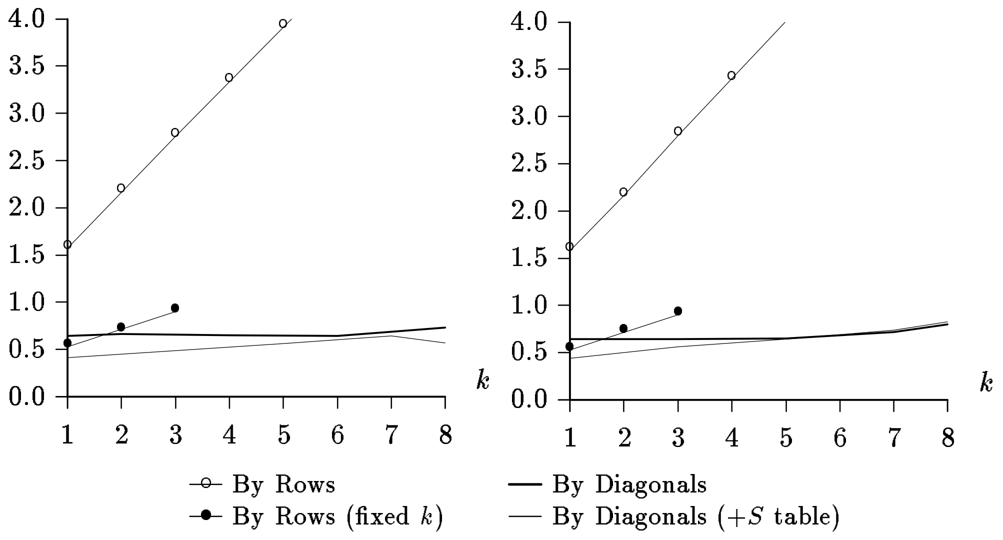


Figure 5.8: Times in seconds for $m = 9$, $n = 1$ Mb. The plot on the left is for random text ($\sigma = 32$), and that on the right is for English text.

### 5.7.1 The Simple Algorithm

The preprocessing phase of this algorithm can be optimized to take $O(\sigma + m \min(m, \sigma))$ time, and it requires $O(\sigma)$ space. The search phase needs $O(n)$ time.

However, this algorithm is limited to the case in which $(m - k)(k + 2) \leq w$. In the RAM model it is assumed $\log_2 n \leq w$, so a machine-independent bound is $(m - k)(k + 2) \leq \log_2 n$.

Since $(m - k)(k + 2)$ takes its maximum value when $k = m/2 - 1$, we can assure that this algorithm can be applied whenever $m \leq 2(\sqrt{w} - 1)$, independently of $k$. That is, we have a linear algorithm for $m = O(\sqrt{\log n})$, for example, $m \leq 9$ for $w = 32$ bits, or $m \leq 14$ for $w = 64$ bits.

Our algorithm shares with Wu and Manber work [WM92a] the NFA model and the idea of using bit-parallelism. However, the parallelization techniques are different. We compare both algorithms in their most simple setups.

A general implementation of the Wu and Manber code needs to use an array of size $k+1$. However, we implemented optimized versions for $k = 1$, 2 and 3. That is, a different code was developed for each $k$ value, in order to avoid the use of arrays and enable the use of machine registers. We show both algorithms (optimized and general). We also show the effect of our simple speed-up heuristic (the $S$ table), running our algorithm with and without that filtration heuristic.

Figure 5.8 shows the results. We show the case $m = 9$ (where we can use the simple algorithm). This test was run on a Sun SparcStation 4 running Solaris 2.3, with 32 Mb of RAM and $w = 32$.

### 5.7.2 Automaton Partitioning

#### 5.7.2.1 Search Cost

The preprocessing time and space complexity of this algorithm is $O(mk\sigma/w)$. Since automaton partitioning gives us some freedom to arrange the cells, we find out now the best arrangement.

First recall that we work only on the active diagonals of the automaton. In Section 4.1 we obtained the expected value for the last active column in the automaton (Eq. (4.5)). This measures active *columns* and we work on active *diagonals.* To obtain the last active diagonal we subtract $k$, to obtain that on average we work on $ke/(\sqrt{\sigma}-e)$ diagonals.

This is because the last active column depends on the error level $k$. Hence, at automaton row $i$ (where only $i$ errors are allowed) the last active column is $lcol(i) = i/(1-e/\sqrt{\sigma})$. Hence, the last active column defines a diagonal line across the automaton whose slope is $1/(1-e/\sqrt{\sigma})$. Figure 5.9 illustrates the situation. All the active states of the automaton are to the left of the dashed diagonal. The number of diagonals affected from the first one (thick line) to the dashed one is $k/(1-e/\sqrt{\sigma})-k$.

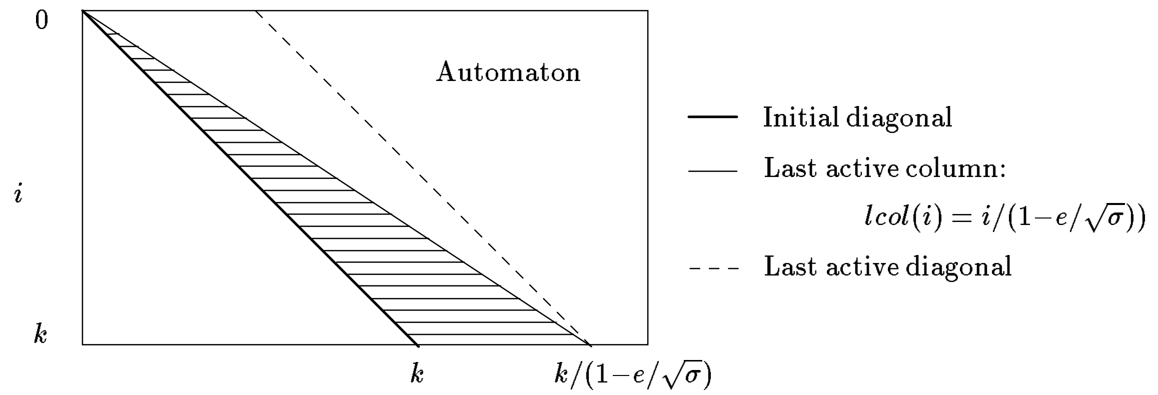


Figure 5.9: Converting active columns to active diagonals. The shaded area represents the active states of the automaton.

Since we pack $(m-k)/J$ diagonals in a single cell, we work on average on $ke/(\sqrt{\sigma}-e) \times J/(m-k)$ d-columns. Each d-column must work on all its $I$ cells. On the other hand, there are only $J$ d-columns. Hence our total complexity is

$$I\ J\ \min\left(1, \frac{ke}{(m-k)(\sqrt{\sigma}-e)}\right)\ n$$

which shows that any choice for $I$ and $J$ is the same for a fixed $IJ$. Since $IJ \approx (m-k)(k+2)/w$ (total number of bits to place divided by the size of the computer word), the final cost expression is independent (up to round-offs) of $I$ and $J$:

$$\min\left(m-k\ ,\ \frac{ke}{\sqrt{\sigma}-e}\right)\ \frac{k+2}{w}\ n \tag{5.6}$$

This formula has two parts. First, for $\alpha < 1-e/\sqrt{\sigma}$, it is $O(k^2n/(\sqrt{\sigma}w))$ time. Second, if the error ratio is high ($\alpha \geq 1-e/\sqrt{\sigma}$), it is $O((m-k)kn/w)$. This last complexity is also the worst case of this algorithm. Recall that in practice the value $e$ should be replaced by 1.09 and the average number of active columns is that of Eq. (4.6).

### 5.7.2.2 Practical Tuning

Since the crude analysis does not give us any clue about which is the optimal selection for $I$ and $J$, we perform a more detailed analysis. The automaton is partitioned into a matrix of $I$ rows and $J$ columns, each cell being a small sub-automaton that stores $\ell_r$ rows of $\ell_c$ diagonals of the complete automaton. Because of the nature of the update formula, we need to store $(\ell_r+1)\ell_c$ bits for each sub-automaton. Thus, the conditions to meet are

$$(\ell_r+1)\ell_c \le w \ , \qquad I = \left\lceil \frac{k+1}{\ell_r} \right\rceil \ , \qquad J = \left\lceil \frac{m-k}{\ell_c} \right\rceil$$

Notice that in some configurations the cells are better occupied than in others, due to round-offs. That is, once we select $\ell_r$ and $\ell_c$, the best possible packing leaves some bits unused, namely $w-(\ell_r+1)\ell_c$.

One could, in fact, try every $I$ and $J$ and pick the configuration with fewer cells. Since we work proportionally to the number of cells, this seems to be a good criterion. Some configurations need more cells than others because, due to round-offs, they use less bits in each computer word (i.e. cell). In the worst possible configuration, $w/2+1$ bits can be used out of $w$, and in the best one all the $w$ bits can be used. It is clearly not possible to use as few as $w/2$ bits or less, since in that case there is enough room to pack the bits of two cells in one, and the above equations would not hold. Hence, the best we can obtain by picking a different $I$ and $J$ is to reduce the number of cells by a factor of 2.

However, by selecting minimal $I$, the possible automata are[3]: ($a$) horizontal ($I=1$), ($b$) horizontal and with only one diagonal per cell ($I=1, \ell_c=1$), or ($c$) not horizontal nor vertical, and with only one diagonal per cell ($I>1, J>1, \ell_c=1$). Those cases can be solved with a simpler update formula (2 to 6 times faster than the general one), since some cases of communication with the neighbors are not present. Moreover, a more horizontal automaton makes the strategy of active columns work better.

This much faster update formula is more important than the possible 2-fold gains due to round-offs. Hence, we prefer to take minimal $I$, i.e.

$$I = \lceil (k+1)/(w-1) \rceil \ , \qquad \ell_r = \lceil (k+1)/I \rceil \ , \qquad \ell_c = \lfloor w/(\ell_r+1) \rfloor \ , \qquad J = \lceil (m-k)/\ell_c \rceil$$

However, the three cases mentioned do not cover ($d$) a purely vertical partitioning, (i.e. $J=1$), which is applicable whenever $2(m-k) \le w$ and has also a simple update formula. The selection for vertical partitioning is $J_v=1$, $\ell_{vc}=m-k$, $\ell_{vr}=\lfloor w/(m-k) \rfloor - 1$, $I_v = \lceil (k+1)/\ell_{vr} \rceil$. Figure 5.10 shows an experimental comparison between ($c$) and ($d$).

The mechanism we use to determine the optimal setup and predict its search cost integrates experimental and analytical results, as follows.

- We experimentally obtain the time that each type of automaton spends per text character (using least squares over real measures). We express those costs normalized so that the cost of the core algorithm is 1.00. These costs have two parts:
  - A base cost that does not depend on the number of cells: ($a$) 1.02, ($b$) 1.13, ($c$) 0.12, ($d$) 1.66.
  - A cost per processed cell of the automaton: ($a$) 1.25, ($b$) 0.83, ($c$) 2.27, ($d$) 1.36.
  - A cost spent in keeping account of which is the last active diagonal: ($a$) 0.68, ($b$) 0.20, ($c$) 1.66. Notice that although at a given text position this work can be proportional to the number of active columns, the amortized cost is $O(1)$ per text position. To see this, consider that at each text character we can at most increment by one the last active

[3]This is true provided we solve the case $k=m-1$ with a simpler algorithm, i.e. the $S$ filter alone.

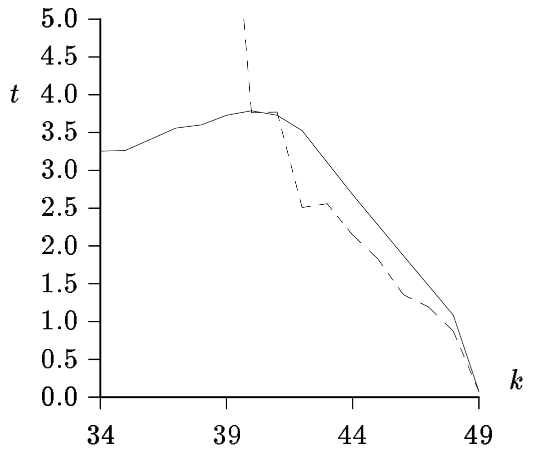


Figure 5.10: Times in seconds for vertical partitioning (dashed line) versus minimal rows partitioning (solid line). We use $m = 50$, $w = 32$, $\sigma = 32$, $n = 1$ Mb, random text and patterns.

column, and therefore no more than $n$ increments and $n$ decrements are possible in a text of size $n$. Hence the correct choice is to consider this cost as independent on the number of cells of the automaton.

- We analytically determine using Eq. (4.6) the expected number of active d-columns.
- Using the above information, we determine whether it is convenient to keep track of the last active column or just modify all columns (normally the last option is better for high error ratios). We also determine which is the most promising partition.

Since this strategy is based on very well-behaved experimental data, it is not surprising that it predicts very well the cost of automaton partitioning and that it selected the best strategy in almost all cases we tried (in some cases it selected a strategy 5% slower than the optimal, but not more).

Finally, notice that the worst case complexity of $O(k(m-k)/w)$ per inspected character is worse than the $O(m)$ of dynamic programming when the pattern length gets large, i.e. $m > w/(\alpha(1-\alpha))$. This ensures that automaton partitioning is better for $m \leq 4w$, which is quite large. In fact, we should also account for the constants involved. The constant for partitioned automata is nearly twice as large as that of dynamic programming, which makes sure that this method is better for $m \leq 2w$. We use therefore a partitioned automaton instead of dynamic programming as our verification engine for potential matches in the sections that follow.

Figure 5.11 shows an experimental comparison between plain dynamic programming, the Ukkonen cutoff variant [Ukk85b] and our partitioned automaton for large patterns. In the worst moment of the partitioned automaton, it is still faster than dynamic programming up to $m = 60$, which confirms our assumptions. As explained at the end of Section 5.9, the peaks in our algorithm are genuine.

#### 5.7.2.3 Improving Register Usage

We finish this section explaining an improvement in the engineering of the algorithm that leads to triplicating the performance in some cases. The improvement is based on better usage of the

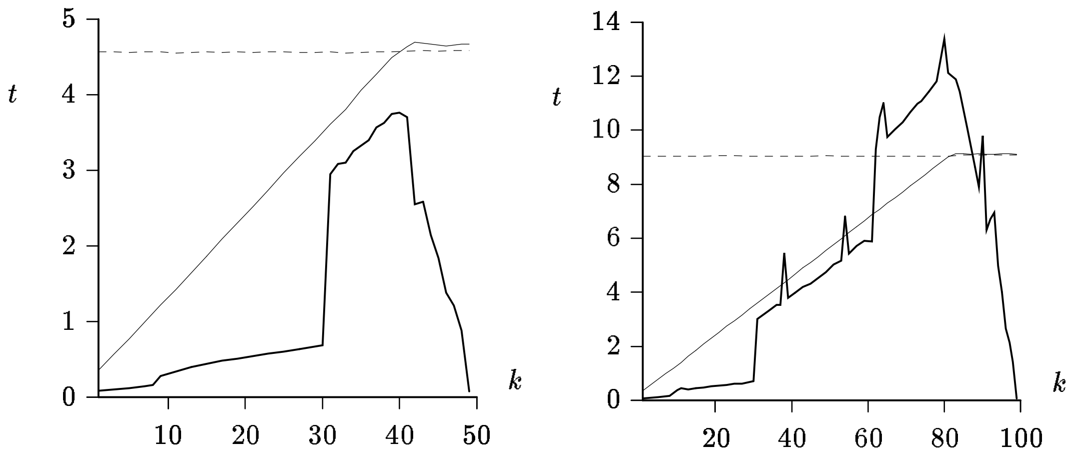


Figure 5.11: Times in seconds for partitioned automaton (thick line) versus dynamic programming (dashed line) and the Ukkonen's improvement (solid thin line). The left plot is for $m = 50$ and the right one for $m = 100$. We use $w = 32$, $\sigma = 32$, $n = 1$ Mb, random text and patterns.

computer registers.

The main difference in the cost between the core algorithm and an horizontally partitioned automaton is that in the first case we can put in a register the machine word which simulates the automaton. This cannot be done in a partitioned automaton, since we use an array of words. The locality of accesses of those words is very low, i.e. if there are $a$ active d-columns, we update for each text character all the words from $D_1$ to $D_a$. Hence, we cannot keep them in registers.

An exception to the above statement is the case $a = 1$. This represents having active only the first cell of the horizontal automaton. We can, therefore, put that cell in a register and traverse the text updating it, until the last diagonal inside the cell becomes active. At that point, it is possible that the second cell will be activated at the next character and we must resume the normal searching with the array of cells. We can return to the one-cell mode when the second cell becomes inactive again.

With this technique, the search cost for a pattern is equal to that of the core algorithm until the second automaton is activated, which in some cases is a rare event. In fact, we must adjust the above prediction formulas, so that the horizontal automata cost the same as the core algorithm (1.00), and we add the above computed cost only whenever their last diagonal is activated. The probability of this event is $f(\ell_c + k, k)$.

This technique elegantly generalizes a (non-elegant) truncation heuristic proposed in earlier versions of this work [BYN98d]. It stated that, for instance, if we had $m = 12$, $k = 1$, better than partitioning the automaton in two we could just truncate the pattern by one letter, use the core algorithm and verify each occurrence. With the present technique we would automatically achieve this, since the last letter will be isolated in the second cell of the horizontal automaton.

Notice that this idea cannot be applied to the case $I > 1$, since in that case we have always more than one active cell. In order to use the technique also for this case, and in order to extend the idea to not only the first cell, we could develop specialized code for two cells, for three cells, and so on, but the effort involved and the complexity of the code are not worth it.

Figure 5.12 shows the improvements obtained over the old version. The better register usage is more noticeable for low error levels (horizontal partitioning).

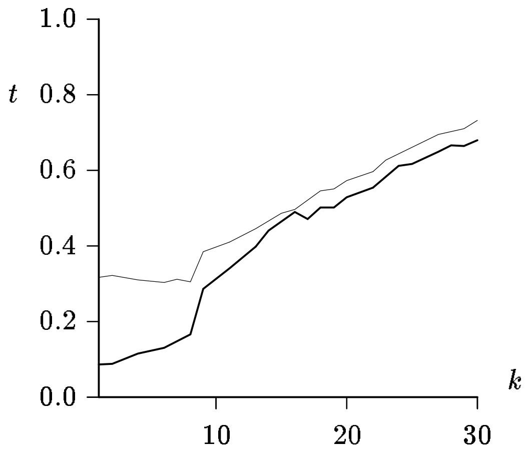


Figure 5.12: Times in seconds for partitioned automata before (thin line) and after (thick line) improving the register usage. We use $m = 60$, $w = 32$, $\sigma = 32$, $n = 1$ Mb, random text and patterns.

### 5.7.3 Pattern Partitioning

#### 5.7.3.1 Search Cost

The cost of pattern partitioning depends on the number $j$ of pieces used. In the worst case each text position must be verified, and since we avoid re-checking the same text, we have the same cost as automaton partitioning. We consider the average case in which follows. This analysis holds only for non-extended patterns.

The minimum number of pieces $j^*$ is given by Eq. (5.5), from which we derive the next equation

$$\left(\frac{m}{j^*} - \frac{k}{j^*}\right)\left(\frac{k}{j^*} + 2\right) = w$$

whose solution (disregarding roundoffs) is

$$j^* = \frac{m - k + \sqrt{(m-k)^2 + wk(m-k)}}{w} \quad = \quad m\ d(w,\alpha) \tag{5.7}$$

where

$$d(w,\alpha) = \frac{1-\alpha}{w}\ \left(1 + \sqrt{1 + w\alpha/(1-\alpha)}\right)$$

As a function of $\alpha$, $d(w,\alpha)$ is convex and is maximized for $\alpha = 1/2\ (1 - 1/(\sqrt{w} - 1))$, where it takes the value $1/(2(\sqrt{w} - 1))$. To give an idea of the reduction obtained over the classical $O(mn)$ cost, this maximum value is 0.11 for $w = 32$ and 0.07 for $w = 64$.

Observe that we discarded the second condition of Eq. (5.5), namely $\lfloor m/j \rfloor > \lfloor k/j \rfloor$. This is because if $\lfloor m/j \rfloor = \lfloor k/j \rfloor$, then $j > m - k$, which implies $\alpha > 1 - 1/(w-1)$. As we show next, this value of $\alpha$ is outside our area of interest (i.e. it is larger than $1 - 1/\sqrt{\sigma}$), except for $\sigma > (w-1)^2$, that is, extremely large alphabets (e.g. $\sigma$ close to 1000 for $w = 32$).

Excluding verifications, the search cost is $O(j^*n)$. For very low error ratios ($\alpha < 1/w$), $j^* = O(m/w)$ and the cost is $O(mn/w)$. For higher error ratios, $j^* = O(\sqrt{mk/w})$ and then the search cost is

$O(\sqrt{mk/w}\ n)$. Both cases can be obviously bounded by $O(mn/\sqrt{w})$. The preprocessing time and storage requirements for the general algorithm are $j$ times those of the simple one.

We consider now the other component of the search cost, that is, the cost to verify the neighborhoods of the text places which match some subpattern. If we simply verify the complete area for the complete pattern, then we have that for each matching piece we have to pay $O(m^2)$ time to verify the neighborhood of the area where it occurred. In Section 4.1 we have proven that the probability that a piece of length $m/j^*$ matches is $O(\gamma^{m/j^*})$, where $\gamma$ is defined in Eq. (4.1) and it holds that $\gamma < 1$ whenever $\alpha < 1 - e/\sqrt{\sigma}$ (notice that $\alpha$ is the same for all the subpatterns). Then, the average cost paid to check all the matches of each piece is $O(m^2\gamma^{m/j^*})$. For this cost (which has to be added up for all the $j^*$ pieces) not to affect the total $O(j^*n)$ search cost, we need $\gamma \leq 1/m^{2j^*/m}$, i.e.

$$\alpha \;\leq\; 1 - \frac{e}{\sqrt{\sigma}}\, m^{\frac{j*}{m-k}} \;=\; 1 - \frac{e}{\sqrt{\sigma}}\, m^{\frac{d(w,\alpha)}{1-\alpha}}$$

which decreases as $m$ grows. Therefore, this method degrades for longer patterns. This is caused mainly because a large pattern is verified whenever any pieces matches. Hence, the verification cost keeps constant while the probability to make the verification increases with $j^*$ (i.e. with $m$). This makes this method to stop working long before the limit $\alpha < 1 - 1.09/\sqrt{\sigma}$ shown in Section 4.1.

However, the hierarchical verification technique presented in Section 4.3.1 does *not* degrade as the pattern grows. In that section (Eq. (4.7)) we show that the total amount of verification work for each piece is $O((m/j*)^2\gamma^{m/j*})$. This is much better than $O(m^2\gamma^{m/j*})$, and in particular it is $O(1)$ whenever $\gamma < 1$, so it does not affect the total search cost of the pieces, even if $m$ grows.

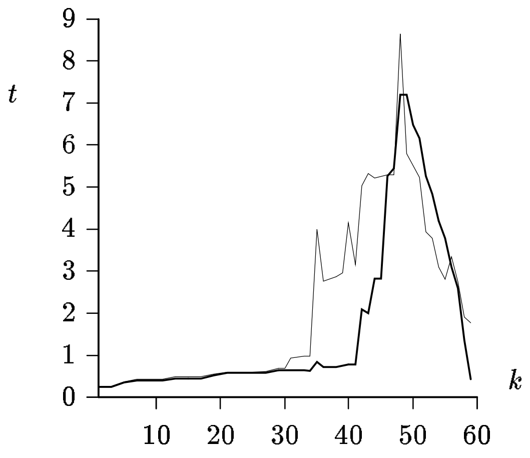
Although when there are few matches (i.e. low error level) the simple and hierarchical verification methods behave similarly, there is an important difference for medium error levels: the hierarchical algorithm is more tolerant to errors. We illustrate this fact in Figure 5.13. As it can be seen, both methods eventually are overwhelmed by the verifications *before* reaching the limit $\alpha = 1 - 1.09/\sqrt{\sigma}$. This is because, as $j$ grows, the cost of verifications $O((m/j^*)^2\gamma^{m/j^*})$ increases. In the case $\sigma = 32$, the theoretical limit is $\alpha^* = 0.83$ (i.e. $k = 50$), while the simple method ceases to be useful for $k = 35$ (i.e. $\alpha = 0.58$) and the hierarchical one works well up to $k = 42$ (i.e. $\alpha = 0.7$). For English text the limit is $\alpha^* = 0.69$, while the simple method works up to $k = 30$ ($\alpha = 0.50$) and the hierarchical one up to $k = 35$ ($\alpha = 0.58$).

It is also noticeable that the hierarchical method works a little harder in the verifications once they become significative (very high error levels). This is because the hierarchy of verifications makes it to check many times the same text area. On the other hand, we notice that the use of partitioned automata instead of dynamic programming for the verification of possible matches is especially advantageous in combination with hierarchical verification, since in most cases we verify only a short pattern, where the automaton is much faster than dynamic programming.

#### 5.7.3.2 Optimal Selection for $j$

It is possible to use just automaton partitioning to solve a problem of any size. It is also possible to use just pattern partitioning, with $j$ large enough for the pieces to be tractable with the kernel algorithm directly (i.e. $j = j^*$).

It is also possible to merge both techniques: partition the pattern into pieces. Those pieces may or may not be small enough to use the kernel algorithm directly. If they are not, search them using automaton partitioning. This has the previous techniques as particular cases.



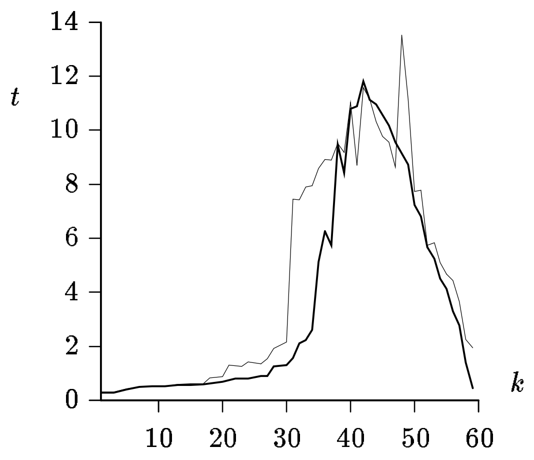


Figure 5.13: Times in seconds for simple (thin line) versus hierarchical (thick line) algorithms for pattern partitioning. We use $m = 60$, $w = 32$, and $n = 1$ Mb. On the left, random text ($\sigma = 32$). On the right, English text.

To obtain the optimal strategy, consider that if we partition in $j$ subpatterns, we must perform $j$ searches with $\lfloor k/j \rfloor$ errors. For $\alpha < 1 - e/\sqrt{\sigma}$, the cost of solving $j$ subproblems by partitioning the automaton is (using Eq. (5.6))

$$\frac{\frac{ke/j}{\sqrt{\sigma}-e}\;(k/j+2)}{w}\;jn \;=\; \frac{ke(k/2+2)}{(\sqrt{\sigma}-e)w}\;n$$

which shows that the lowest cost is obtained with the largest $j$ value, and therefore $j = j^*$ is the best choice.

However, this is just an asymptotic result. In practice the best option is more complicated due to simplifications in the analysis, constant factors, and integer roundoffs. For instance, a pattern with 4 pieces can be better searched with two horizontal automata of size $(I = 1, J = 2)$ than with four simple automata (especially given the improvements of Section 5.7.2.3). The cost of each automaton depends heavily on its detailed structure. Therefore, to determine the best option in practice we must check all the possible $j$ values, from 1 to $j^*$ and predict the cost of each strategy. This cost accounts for running $j$ automata of the required type (which depends on $j$), as well as for the cost to verify the potential matches multiplied by their probability of occurrence (using Eq. (4.1)).

### 5.7.4 Superimposition

#### 5.7.4.1 Optimizing the Amount of Superimposition

Suppose we decide to superimpose $r$ patterns in a single search. We are limited in the amount of this superimposition because of the increase in the error level to tolerate, with the consequent increase in the cost of verifications. We analyze now how many patterns we can superimpose.

As shown in Section 4.1 (Eq. (4.1)), the probability of a given text position matching a random pattern is $O(\gamma^m)$, where $\gamma$ depends on $\alpha$ and $\sigma$. This cost is exponentially decreasing with $m$ for $\alpha < 1 - e/\sqrt{\sigma}$, while if this condition does not hold the probability is very high.

In this formula, $1/\sigma$ stands for the probability of a character crossing a horizontal edge of the automaton (i.e. the probability of two random characters being equal). To extend this result, we notice that we have $r$ characters on each edge now, so the above mentioned probability is $1-(1-1/\sigma)^r \approx r/\sigma$. The (pessimistic) approximation is tight for $r << \sigma$. We use the approximation because in practice $r$ will be quite modest compared to $\sigma$.

Hence, the value of $\gamma$ when superimposing $r$ patterns (which we call $\gamma'$ to keep unchanged the old $\gamma$ value) is

$$\gamma' \;=\; \left(\frac{r}{\sigma\alpha^{\frac{2\alpha}{1-\alpha}}(1-\alpha)^2}\right)^{1-\alpha} \;=\; r^{1-\alpha}\,\gamma \tag{5.8}$$

and therefore the new limit for $\alpha$ is

$$\alpha < 1 - e\sqrt{\frac{r}{\sigma}}$$

or alternatively the limit for $r$ (i.e. the maximum amount of superimposition $r^*$ that can be used given the error level) is

$$r^* \;=\; \frac{\sigma\,(1-\alpha)^2}{e^2}$$

which for constant error level is $O(\sigma)$ independent on $m$. This is not the only restriction on $r$, because we must check all the $r$ superimposed patterns in the area and therefore the verifications cost more. However, as shown in Section 4.3.2, our average verification cost is independent on $r$ provided we search them hierarchically. Thanks to this and the other hierarchical verification mechanism (of Section 4.3.1) we can superimpose more patterns than if using simple verification. This translates into better performance everywhere, not only when the error level is becoming high.

Considering the above limit, the total search cost becomes $1/r^* = O(1/(\sigma\,(1-\alpha)^2))$ times that of pattern partitioning. For instance, if we partition in $j^*$ pieces (so that they can be searched with the core algorithm), the search cost becomes

$$O\left(\frac{m\ d(w,\alpha)}{\sigma(1-\alpha)^2}\ n\right)$$

which for $\alpha \le 1/w$ is $O(mn/(w\sigma))$, and for higher error level becomes $O(\sqrt{mk/(w\sigma)}\ n)$ (this is because $1-\alpha$ is lower bounded by $e/\sqrt{\sigma}$). Again, a general bound is $O(mn/\sqrt{w\sigma})$.

A natural question is for which error level can we superimpose all the $j^*$ patterns to perform just one search, i.e. when $r^* = j^*$ holds. That is

$$m\ d(w,\alpha) = \frac{\sigma(1-\alpha)^2}{e^2}$$

whose approximate solution is

$$\alpha \;<\; \alpha_1 \;=\; 1 - \frac{e^2 m}{\sigma\sqrt{w}} \tag{5.9}$$

where as always we must replace $e$ by 1.09 in practice. As we see in the experiments, this bound is pessimistic because of the roundoff factors which affect $j^*$ for medium-size patterns.

Notice that superimposition stops working when $r^* = 1$, i.e. when $\alpha = 1 - e/\sqrt{\sigma}$. This is the same point when pattern partitioning stops working. We show in Figure 5.14 the effect of superimposition on the performance of the algorithm and its tolerance to the error level. As we see in Section 5.9, we achieve almost constant search time until the error level becomes medium. This is because we automatically superimpose as much as possible given the error level.

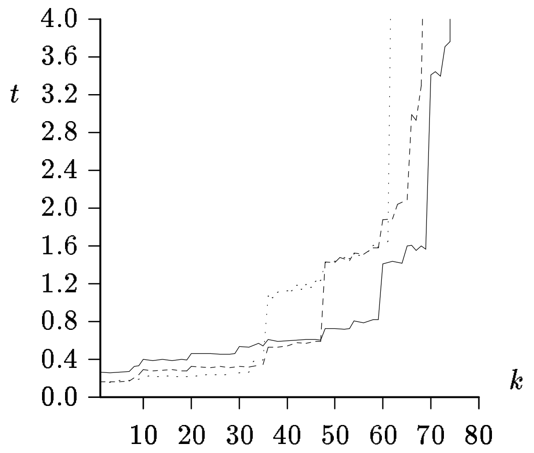


Figure 5.14: Times in seconds for superimposed automata. Superimposition is forced to $r = 2$ (solid line), 4 (dashed line) and 6 (dotted line). The larger $r$, the faster the search but it stops working for lower error levels. We use $m = 100$, $w = 32$, and $n = 1$ Mb and random text and patterns with $\sigma = 32$.

#### 5.7.4.2 Optimal Grouping and Aligning

Two final aspects allow further optimization. A first one is that it is possible to try to form the groups so that the patterns in each group are similar (e.g. they are at small edit distance among them, or they share letters at the same position). This would decrease the probability of finding spurious matches in the text. A possible disadvantage of this heuristic is that since the subpatterns are not contiguous we cannot simply verify whether their concatenation appears, but we have to check if any of the corresponding leaves of the tree appears. The probability that the concatenation appears is much lower.

A second one is that, since we may have to prune the longer subpatterns of each group, we can determine whether to eliminate the first or the last character (the pattern lengths differ at most by one), using the same idea of trying to make the patterns as similar as possible.

None of these heuristics have been tested yet.

## 5.8 Combining All the Techniques

At this point, a number of techniques have been described, analyzed and optimized. They can be used in many combinations for a single problem. A large pattern can be split into one or more subpatterns (the case of "one" meaning no splitting at all). Those subpatterns can be small enough to be searched with the kernel algorithm or they can be still large and need to be searched with a partitioned automaton. Moreover, we can group those automata (simple or partitioned) to speed up the search by using superimposition.

The analysis helped us to find more efficient techniques and to determine the area where each technique can be used. However, a number of questions still arise. Which is the correct choice to split the pattern versus the size of the pieces? Is it better to have fewer pieces or smaller pieces? How does the superimposition affect this picture? Is it better to have more small pieces and superimpose more pieces per group or is it better to have larger pieces and smaller groups?

We study the optimal combination in this section. We begin showing the result of a theoretical analysis and then explain the heuristic we use.

### 5.8.1 A Theoretical Approach

The analysis recommends using the maximal possible superimposition, $r = r^*$, to reduce the number of searches. As proved in Section 5.7.3.2, it also recommends to use the maximal $j = j^*$. This gives the following combined (simplified) average complexity for our algorithm, illustrated in Figure 5.15:

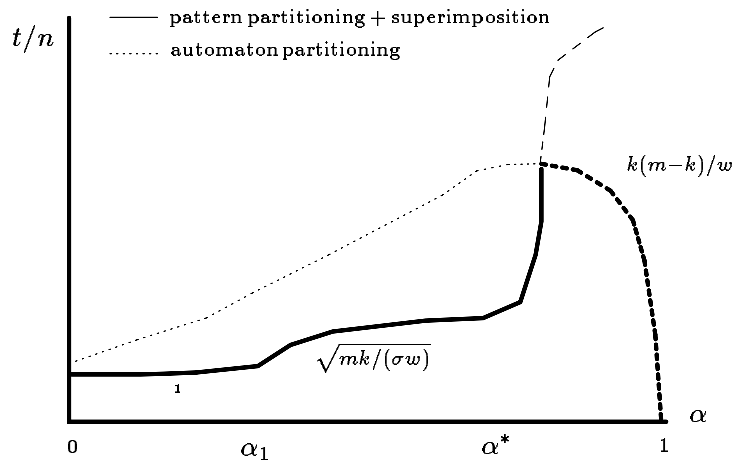


Figure 5.15: The simplified complexity of our algorithm.

- If the problem fits in a machine word (i.e. $(m-k)(k+2) \leq w$), the core algorithm is used at $O(n)$ average and worst-case search cost.
- If the error level is so low that we can cut the pattern in $j^*$ pieces and superimpose all them (i.e. $\alpha < \alpha_1$, Eq. (5.9)) then superimposed automata gives $O(n)$ average search cost.
- If the error level is not so low but it is not too high (i.e. $\alpha < \alpha^*$, Eq. (4.2)), then use pattern partitioning in $j^*$ parts, to obtain near $O(\sqrt{mk/(w\sigma)}\ n)$ average search cost.
- If the error level is too high (i.e. $\alpha > \alpha^*$) we must use automaton partitioning at $O(k(m-k)n/w)$ average and worst-case search cost.

On the other hand, the worst-case search cost is $O(k(m-k)/w\ n)$ in all cases. This is the same worst-case cost of the search using the automaton. This is because we use such an automaton to verify the matches, and we never verify a text position twice with the same automaton. We keep the state of the search and its last text position visited to avoid backtracking in the text due to overlapping verification requirements. This argument is valid even with hierarchical verification.

### 5.8.2 A Practical Heuristic and a Searching Software

Clearly the theoretical analysis alone is insufficient at this point. The results are asymptotic and do not account for many details which are important in practice, such as roundoffs and constant factors.

The real costs are so complex that the best way to find the optimal combination relies on trying all the possible values of $j$, from 1 to $j^*$ and for $r$, from 1 to $j$. For each possible value of $r$ and $j$, we compute the predicted cost of performing the $\lceil j/r \rceil$ searches with simple or partitioned automata as explained in Section 5.7.2.2. We also account for the probability of matching the (possibly superimposed) automata in the text, which is obtained from Eq. (5.8), as well as the cost of such verification. This is an inextricable mix of theoretical and empirical results. This prediction algorithm costs $O(k^2)$, which is quite modest. Its outcome is not only the recommended combination of techniques to use, but also the expected cost of the search.

This heuristic algorithm has been implemented as a software system, which is publicly available from `http://www.dcc.uchile.cl/~gnavarro/pubcode`. This software uses the techniques in an optimal way, but it also allows forcing the use of any combination for test purposes. It also allows forcing or avoid using the twist mentioned in Section 5.2.1. It reports the combination of parameters used, the time spent in the search and the number of matches found. It can optionally print those matches. Currently the software needs to be provided with the value of $\sigma$. We plan in the future a self-adjusting feature that makes it able not only of determine the type of text it is in, but also to change the strategy if the selected combination proves bad.

## 5.9 Experimental Comparison

In this section we experimentally compare our combined heuristic against the fastest previous algorithms we are aware of. Since we compare only the fastest algorithms, we leave aside [Sel80, Ukk85a, GP90, LV89, Tak94, Wri94, WM92a, Ukk85b, ST95], which are not competitive in the range of parameters that we study here. Our algorithm is shown using and not using speed-up of Section 5.2.1, since it could be applied to many other algorithms as well (but generally not to filtration algorithms).

We tested random patterns against 10 Mb of random text. We test $\sigma = 32$ and English text. Each data point was obtained by averaging the Unix's user time over 10 trials. We present all the times in tenths of seconds per megabyte of text.

The algorithms included in this comparison follow (in alphabetical code order). More complete explanations on these algorithms can be found in Chapter 3, while we emphasize here their operative details. Notice that we are not including other algorithms developed in this thesis[4], these are considered later in their respective chapters.

**Agrep** [WM92b] is a widely distributed exact and approximate search software oriented to natural language text. It is limited (although not intrinsically) to $m \leq 32$ and $k \leq 8$.

**BM** is a filter based on applying a Boyer-Moore-type machinery [TU93]. The code is from J. Tarhio.

**BPM** (bit-parallel matrix) is a recent work [Mye98] based on the bit-parallel simulation of the dynamic programming matrix. The code is from G. Myers and has different versions for one and for multiple machine words.

**Count** is a counting filter proposed in [JTU96], which slides a window over the text counting the number of letters in the text window that are present in the pattern. We use our own variant

[4]Notice that we include some algorithms such as Counting and DFA which, although not created in this thesis, we have studied more in depth and implemented more efficiently.

(window of fixed size, see Section 6.2). In that section we also present an improvement over this technique.

**CP** is the column partitioning algorithm (kn.clp) of [CL92], which computes only the places where the value of the dynamic programming matrix does not change along each column. The code is from W. Chang.

**DFA** converts the NFA into a deterministic automaton which is computed in lazy form. The algorithm is proposed in [Kur96] and studied more in detail in Section 6.4. The code is ours.

**EP** (exact partitioning) is the filtering algorithm proposed in [WM92a] which splits the pattern in $k+1$ pieces and searches them using a Boyer-Moore multipattern algorithm, as suggested in [BYP96]. The code is ours and uses an extension of the Sunday [Sun90] algorithm (an improvement over this code is given in Section 6.1).

**Four-Russians** applies a Four Russians technique to pack many automaton transitions in computer words. The code is from the authors [WMM96], and is used with $r=5$ as suggested in their paper ($r$ is related with the size of the Four Russians tables).

**NFA - NFA/NS** is our combined heuristic, with and without the speed-up technique.

Figure 5.16 shows the results for random text with $\sigma = 32$. As it can be seen, our algorithm is more efficient than any other when the problem fits in a single word ($m = 9$), except for low error level, where EP is unbeaten. For very low error level our algorithm is also beaten by BM. For longer patterns, our algorithm is the fastest one up to shortly after $\alpha = 1/2$. Again, EP is the exception, since it is faster up to $\alpha = 1/3$ approximately. For $\alpha > 1/2$, BPM is the fastest one, except when the pattern is longer than $w$ letters and the error level is high. In this final case, 4-Russians is wins.

Figure 5.17 shows the results for English text. The results are similar but the allowed error ratios are reduced: our algorithm is the fastest up to $\alpha = 1/3$ approximately, except for EP which is faster for $\alpha \leq 1/5$. Agrep is also very efficient for low error levels, quite close to EP. The strange behavior for Agrep in the case $m = 9$ occurs because as soon as it finds a match in the line it reports the line and abandons the search of that line, hence improving for very high error ratios.

Finally, Figure 5.18 shows the results for long patterns and fixed error level. The results show that for long patterns our algorithm and BPM are the fastest if the error level is not too high. For low error levels the algorithm EP is better, but it degrades as $m$ grows.

The reader may be curious about the strange behavior of some of the curves in our algorithms. Those are not caused by statistical deviations in the tests but are due to integer round-offs, which are intrinsic to our algorithms. For instance, if we had to use pattern partitioning to split a search with $m = 30$ and $k = 17$, we would need to search four subpatterns, while for $k = 18$ we need just three. As another example, consider automaton partitioning for $m = 20$ and $k = 13$, 14 and 15. The number of cells to work on ($IJ$) change from four to three and then to five. The use of the smart heuristic eliminates most of those peaks, but some remain.

Another possible concern for the reader is how the results of the experiments differ from one platform to another. Is it possible that algorithms faster than others become slower on another machine? Our experience is that in general the results can be projected to other machines with no changes. We have made the same experiments on a Sun SparcClassic with 16 Mb of RAM running SunOS 4.1.3 and on an Intel 486 DX of 100 MHz with 16 Mb of RAM running Linux. The only difference occurs with respect to algorithms that require large amounts of memory (e.g. [WMM96, Nav97b]), which obviously benefit benefit from larger RAMs.

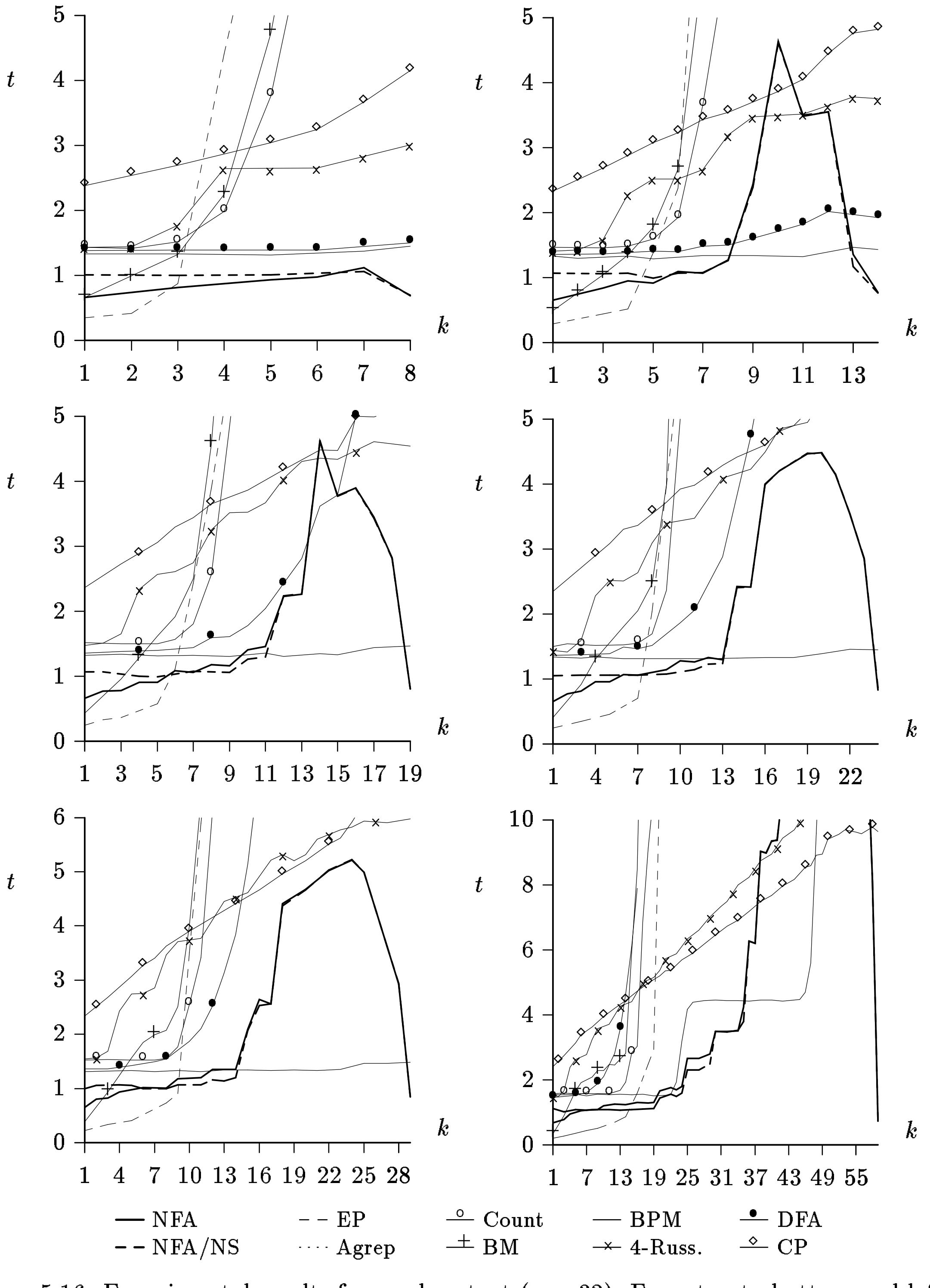


Figure 5.16: Experimental results for random text ($\sigma = 32$). From top to bottom and left to right, $m$ =9, 15, 20, 25, 30 and 60.

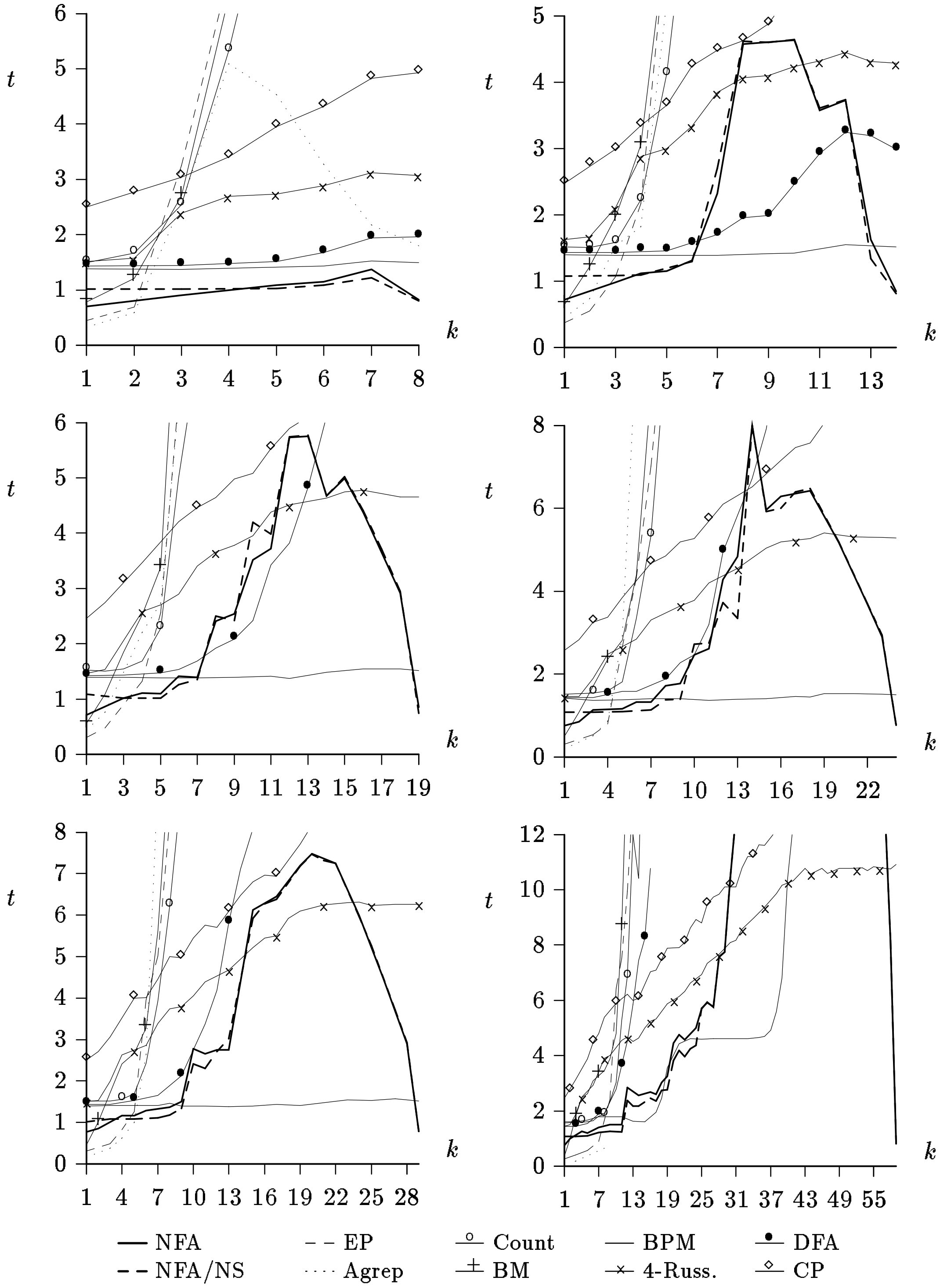


Figure 5.17: Experimental results for English text. From top to bottom and left to right, $m$ =9, 15, 20, 25, 30 and 60.

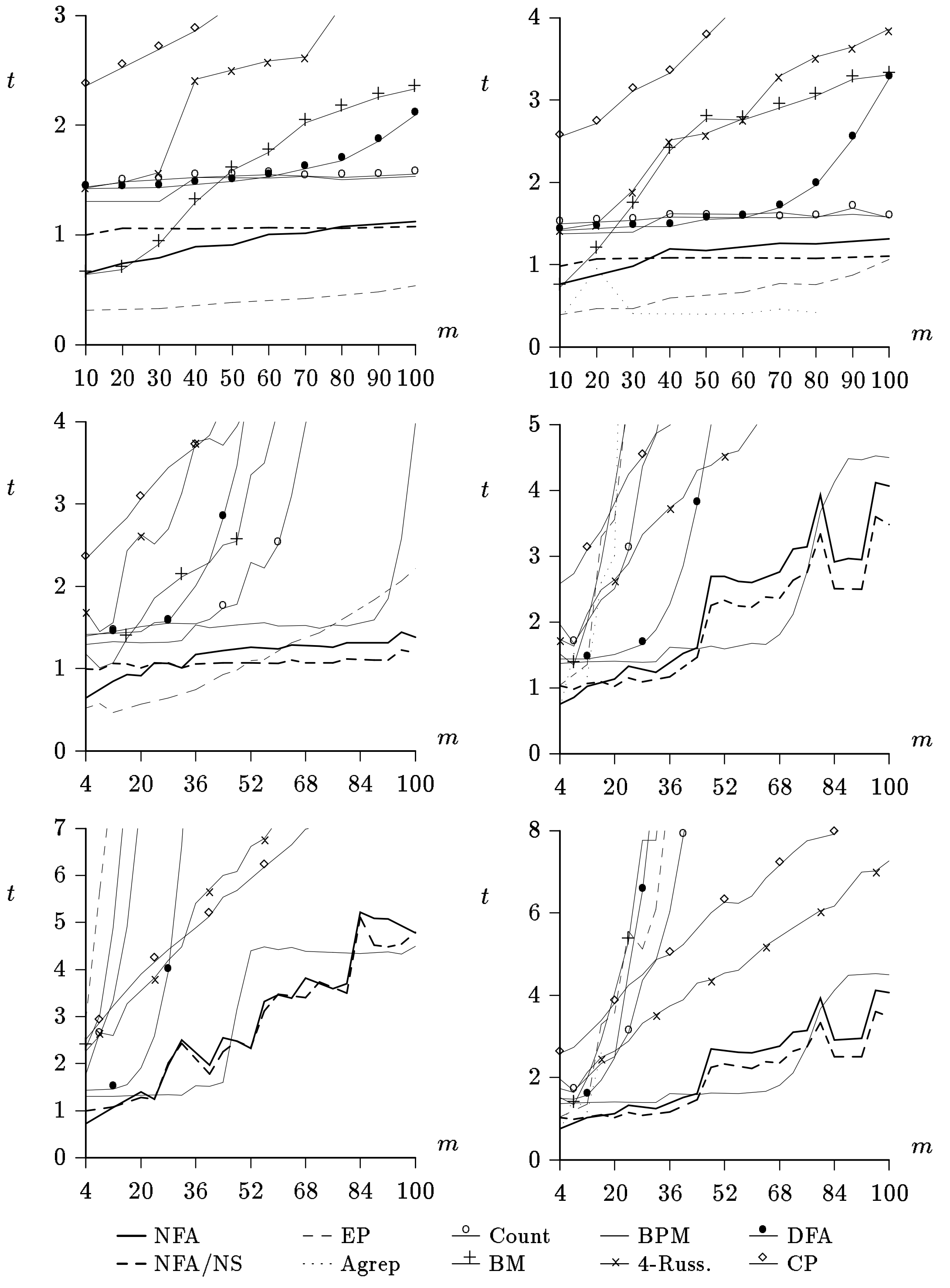


Figure 5.18: Experiments for long patterns. On the left, random text ($\sigma = 32$), on the right, English text. From top to bottom, the plots are for $\alpha = 0.1$, $\alpha = 0.25$ and $\alpha = 0.5$.

# Chapter 6

# Filtering and Automata Algorithms

We present a number of different filtration and automata-based approaches in this chapter. The general idea of filters is to quickly obtain sufficient conditions to discard large text areas so that more expensive algorithms need to be run only on the areas that cannot be discarded. The idea of the automaton approach is to convert it to deterministic and reduce the memory requirements. The ideas presented in this chapter have been published in [NBY98d, Nav97a, NR98b, Nav97b], and some preliminary results can be also found in [BYN96b, BYN96a, BYN98d].

## 6.1 Reduction to Exact Search

In this section we implement a filter proposed in [BYP96] to which little attention was paid before. The result is the fastest known algorithm for approximate string matching. This algorithm, however, can only be used for low error levels. By using a new algorithm to verify potential matches and a new optimization technique for biased texts (such as English), the algorithm becomes the fastest one for medium error levels too. This includes most of the interesting cases in this area, and therefore the result is the fastest algorithm for most cases of interest.

### 6.1.1 The Original Algorithm

The idea of this algorithm is explained in Section 3.1.3. As explained there, the original presentation is from [WM92a], who state the following lemma:

**Lemma:** If a pattern is partitioned in $k+1$ pieces, then at least one of the pieces can be found with no errors in any approximate occurrence of the pattern.

This property is easily verified by considering that $k$ errors cannot alter all the $k+1$ pieces of the pattern, and therefore at least one of the pieces must appear unaltered. In fact it is a particular case of our Partitioning Lemma proved in Section 4.2. Since in this case $j = k+1$, the pieces are searched with $\lfloor k/(k+1) \rfloor = 0$ errors.

This reduces the problem of approximate string searching to a problem of multipattern exact search plus verification of potential matches. That is, we split the pattern in $k+1$ pieces and search all them in parallel with a multipattern exact search algorithm. Each time we find a piece in the text, we verify a neighborhood to determine if the complete pattern appears.

In the original proposal [WM92a], a variant of the Shift-Or algorithm was used for multipattern search. To search $r$ patterns of length $m'$, this algorithm is $O(rm'n/w)$. Since in this case $r = k+1$ and $m' = \lfloor m/(k+1) \rfloor$, the search cost is $O(mn/w)$, which is the same cost for exact searching using Shift-Or. Later, in [BYP96], the use of a multipattern extension of an algorithm of the Boyer-Moore (BM) family was proposed. However, no more attention was paid to this idea until this thesis.

In this work we have implemented the above idea. To select the BM algorithm to extend to multipattern search, we have considered that in practice there would be a few, short pieces to search. We have selected to extend the Sunday [Sun90] algorithm. The extension is as follows.

We split the pattern in pieces of length $\lfloor m/(k+1) \rfloor$ and $\lceil m/(k+1) \rceil$ and form a trie with the pieces. We also build a pessimistic $d$ table with all the pieces (the longer pieces are pruned to build this table). This table stores, for each character, the smallest shift allowed among all the pieces. Now, at each text position we enter in the trie using the text characters from that position on. If we end up in a leaf, we found a piece, otherwise we did not. In any case, we use the $d$ table to shift to the next text position. This simple idea works very well.

We consider verifications now. Suppose we find at $T_{..i}$ the end of a match for the subpattern ending at $P_{..j}$. Then, the potential match must be searched in the area $T_{i-j+1-k..i-j+1+m+k}$, an $(m+2k)$-wide area. This checking must be done with an algorithm resistant to high error levels, such as dynamic programming.

This algorithm is the fastest one in practice when the total number of verifications triggered is low enough, in which case the search cost is $O(kn/m) = O(\alpha n)$ in the best case (this is because the pieces are of length $m/(k+1) \approx 1/\alpha$). We find out now when the total amount of work due to verifications is not higher.

An exact pattern of length $\ell$ appears in random text with probability $1/\sigma^\ell$. In our case, this is $1/\sigma^{\lfloor m/(k+1) \rfloor} \approx 1/\sigma^{1/\alpha}$. Since the cost to verify a potential match using dynamic programming is $O(m^2)$, and since there are $k+1 \approx \alpha m$ pieces to search, the total cost for verifications is $m^3\alpha/\sigma^{1/\alpha}$. This cost must be $O(\alpha)$ so that it does not affect the total cost of the algorithm. This happens for $\alpha < 1/(3\log_\sigma m)$. On English text we found empirically the limit $\alpha < 1/5$ for moderate size patterns.

Compared to the original proposal of [WM92a], the use of the Sunday algorithm is less flexible because it cannot search for extended patterns. However, in [NR98a] the Sunday algorithm was replaced with another one based on bit-parallel suffix automata. The resulting algorithm has almost the same performance and is able to search some extended patterns. However, we focus in simple patterns in this section and keep using the Sunday algorithm.

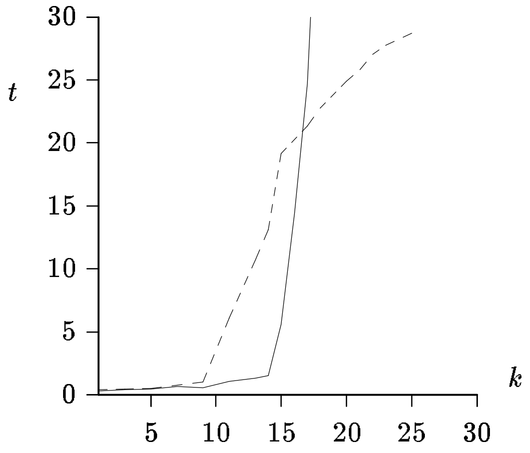
### 6.1.2 Applying Hierarchical Verification

The hierarchical verification technique presented in Section 4.3.1 is useful here. The idea is to try to quickly determine that the match of a small piece is not in fact part of a complete match. Hence, instead of verifying the complete area of interest we perform a hierarchical verification with longer and longer pieces of the pattern that contain the matching piece.

If we use the hierarchical verification algorithm, the analysis of Section 4.3.1 (Eq. (4.8)) shows that the verification cost per piece is $\ell^2/\sigma^{\lfloor \frac{m}{k+1} \rfloor}$, where $\ell = m/(k+1) \approx 1/\alpha$. Since there are $k+1 \approx \alpha m$ pieces to search, the total cost for verifications is $m/(\alpha\sigma^{1/\alpha})$. This cost must be $O(\alpha)$ so that it does not affect the total cost of the algorithm. This happens for

$$\alpha < \frac{1}{\log_\sigma m + 2\log_\sigma(1/\alpha)} = \frac{1}{\log_\sigma m + \Theta(\log_\sigma \log_\sigma m)}$$

Figure 6.1 illustrates the improvement obtained. As it can be seen, on random text the hierarchical verification works well up to $\alpha = 1/2$, while simple verification works well up to $\alpha = 1/3$. On the other hand, after that point the verifications cost much more than in the simple method. This is because of the hierarchy of verifications which is carried out for most text positions when the error level is high. On the other hand, it is hard to improve the barrier of $\alpha < 1/2$ with this method, since at this point we are searching for single characters and performing a verification each time some of the characters is found in the text (which is very frequent).



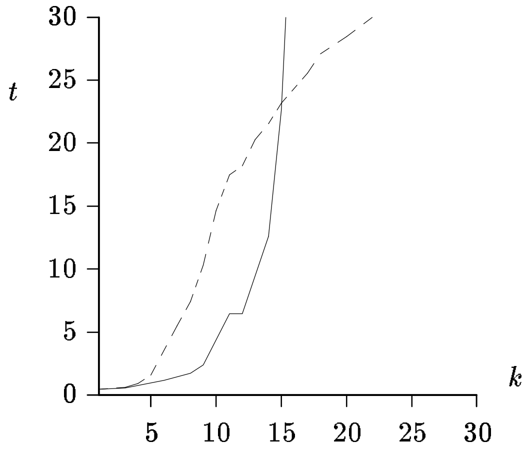


Figure 6.1: The hierarchical (solid line) versus the simple (dashed line) verification technique. We use $m = 60$ and show random (left, $\sigma = 32$) and English text (right). We show the time in seconds for $n = 10$ Mb of text.

### 6.1.3 Optimizing the Partition

When splitting the pattern, we are free to determine the $k+1$ pieces as we like. This is a consequence of our general version of the Partitioning Lemma (Section 4.2), and can be used to minimize the expected number of verifications when the letters of the alphabet do not have the same probability of occurrence (as in English text).

For example, imagine that $Pr('e') = 0.3$ and $Pr('z') = 0.003$. Then, if we search for "eeez" it is better to partition it as "eee" and "z" (with probabilities 0.0027 and 0.003 respectively) rather than "ee" and "ez" (with probabilities 0.09 and 0.0009 respectively). More generally, assuming that the probability of a sequence is the product of the individual letter probabilities[1], we want a partition that minimizes the sum of the probabilities of the pieces (which is directly related to the number of verifications to perform).

We present now a dynamic programming algorithm to optimize the partition of $P_{1..m}$. Let $R[i,j] = \prod_{r=i+1}^{j} Pr(P_r)$ for every $0 \leq i \leq j \leq m$. It is easy to see that $R$ can be computed in $O(m^2)$ time since $R[i,j] = R[i,j-1] \times Pr(P_j)$. Using $R$ we build two matrices, namely

- $SP[i,k]$ = sum of the probabilities of the pieces in the best partition for $P_{i+1..m}$ with $k$ errors.
- $C[i,k]$ = where the next piece must start in order to obtain $SP[i,k]$.

[1]Although we are using a model of individual letters (i.e. 0-order Markov chain), we can easily extend it to a higher order model (e.g. considering probabilities of pairs of letters).

This takes $O(m^2)$ space. The algorithm of Figure 6.2 computes the optimal partition in $O(m^2k)$ time.

```
                /* building $R$ */
for ($i=0; i \le m; i++$) $R[i,i] = 1.0$;
for ($d=1; d \le m; d++$)
    for ($i=0; i \le m-d; i++$)
        $R[i,i+d] = Pr(P_{i+1}) \times R[i+1,i+d]$;
                /* computing $SP$ and $C$ */
for ($i=0; i < m; i++$)
    $SP[i,0] = R[i,m]$;
    $C[i,0] = m$;
for ($r=1; r \le k; r++$)
    for ($i=0; i < m-r; i++$)
        $SP[i,r] = \min_{j \in i+1..m-r}(R[i,j] + SP[j,r-1])$;
        $C[i,r] = j$ that minimizes the expression above;
```

Figure 6.2: Dynamic programming algorithm to optimize the partition of the pattern.

The final probability of verification is $SP[0,k]$ (note that we can use it to estimate the real cost of the algorithm in runtime, before running it on the text). The pieces start at $\ell_0 = 0$, $\ell_1 = C[\ell_0, k]$, $\ell_2 = C[\ell_1, k-1]$, ..., $\ell_k = C[\ell_{k-1}, 1]$.

As we presented the optimization, the obtained speedup is very modest and even counterproductive in some cases. This is because we consider only the probability of verifying. The search times of the extended Sunday algorithm degrades as the length of the shortest piece is reduced, as it happens in an uneven partition. We consider in fact a cost model which is closer to the real search cost. We optimize

$$\frac{1}{\text{minimum length}} + Pr(\text{verifying}) \times m^2$$

Figure 6.3 shows experimental results comparing the normal versus the optimized partitioning algorithms. We repeated this experiment 100 times because of its very high variance. This experiment is only run on English text since it has no effect on random text. Both cases use the original verification method, not the hierarchical one. As it can be seen, the achieved improvements are especially noticeable in the intermediate range of errors.

### 6.1.4 Experimental Comparison

In this section we experimentally compare the old and new algorithms against the fastest algorithms we are aware of. These are explained in detail in Section 5.9.

We tested random text with $\sigma = 32$, and English text. Each data point was obtained by averaging the Unix's user time over 50 trials on 10 megabytes of text. We present all the times in tenths of seconds per megabyte. From the algorithms described in Section 5.9 we only include those that proved to be the best ones: EP (i.e. the original version of this algorithm), NFA (i.e. the algorithm of Chapter 5) and BPM. The improved algorithm of this section is labeled **HEP** (for "hierarchical exact partitioning").

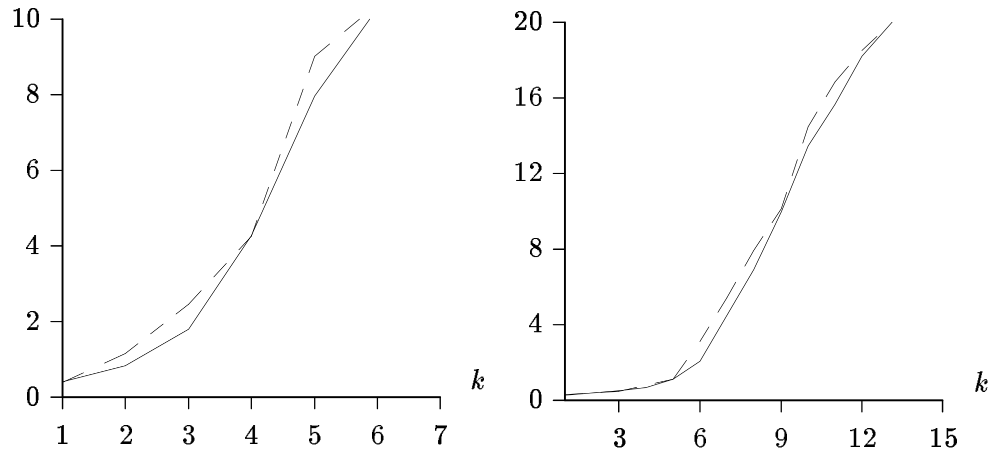


Figure 6.3: The optimized (solid line) and the normal splitting (dashed line), for $m = 10$ and 30 on 10 Mb of English text.

On English text we add two extra algorithms: Agrep and a version of our algorithm that includes the splitting optimization. On English text the code "HEP" corresponds to our algorithm with hierarchical verification and splitting optimization, while "HEP/NO" shows hierarchical verification and no splitting optimization.

As seen in Figure 6.4, for $\sigma = 32$ the new algorithm is more efficient than any other for $\alpha < 1/2$, while for English text it is the fastest for $\alpha < 1/3$. Notice that although Agrep is normally faster than EP (i.e. the original version of this technique), we are faster than Agrep with the hierarchical verification, and the splitting optimization improves a little over this.

Figure 6.5 shows the results for long patterns and fixed error level. For very low error level ($\alpha = 0.1$) our new algorithm improves a little over EP, although for natural language Agrep is the fastest for $m \geq 50$. For low error level ($\alpha = 0.25$) the new algorithm HEP is works much better than the old EP, and becomes the fastest even where EP is not even competitive. Hence, although this algorithm also degrades as $m$ grows, it is much more resistant to the pattern length.

### 6.1.5 Extensions

As explained, this algorithm can reasonably handle extended patterns if the BNDM search algorithm of [NR98a] is used instead of the Sunday extension. Figure 6.6 shows the relative performance of BNDM with respect to Sunday. The three curves correspond to the same algorithm (without hierarchical verification or splitting optimization) where the multipattern search is implemented with BNDM, Sunday or WM (i.e. the first proposal of [WM92a]).

As it can be seen, Sunday is (almost always) better but BNDM is reasonably competitive and more flexible (moreover, its performance does not degrade significantly if classes of characters are allowed, see [NR98a, NR98b]).

We can adapt the filter to other distance functions quite easily. Different costs of the operations can be accommodated by simply determining the minimum number of operations $k$ necessary to reach the allowed error level. We can allow other operations as well, although some of them require more care. For instance, if we allow transpositions we have that a single operation can alter two pattern pieces, and therefore we must split the pattern in $2k + 1$ pieces instead of $k + 1$.

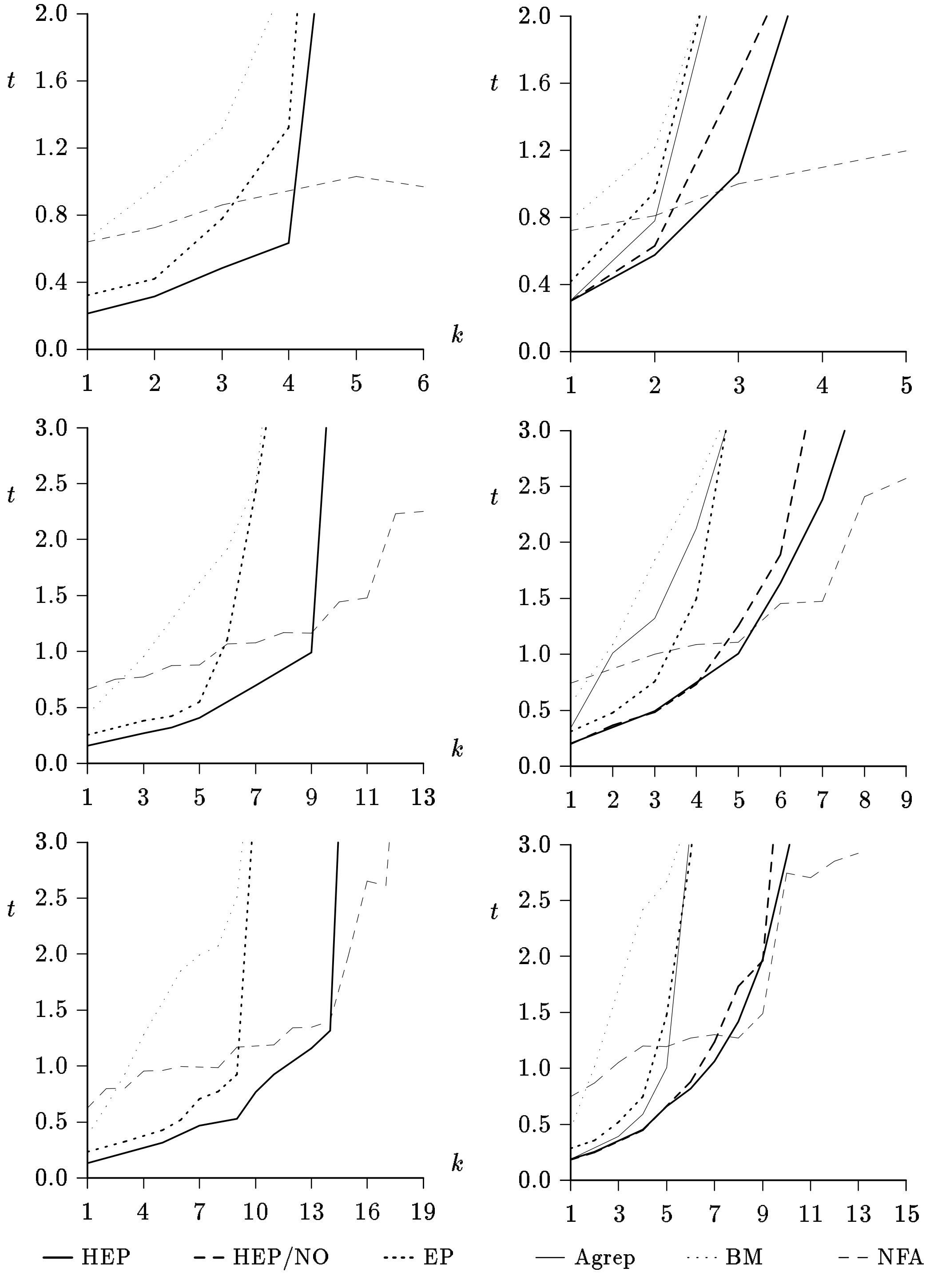


Figure 6.4: Experimental results for random ($\sigma = 32$, left) and English text (right). From top to bottom $m$ =10, 20 and 30.

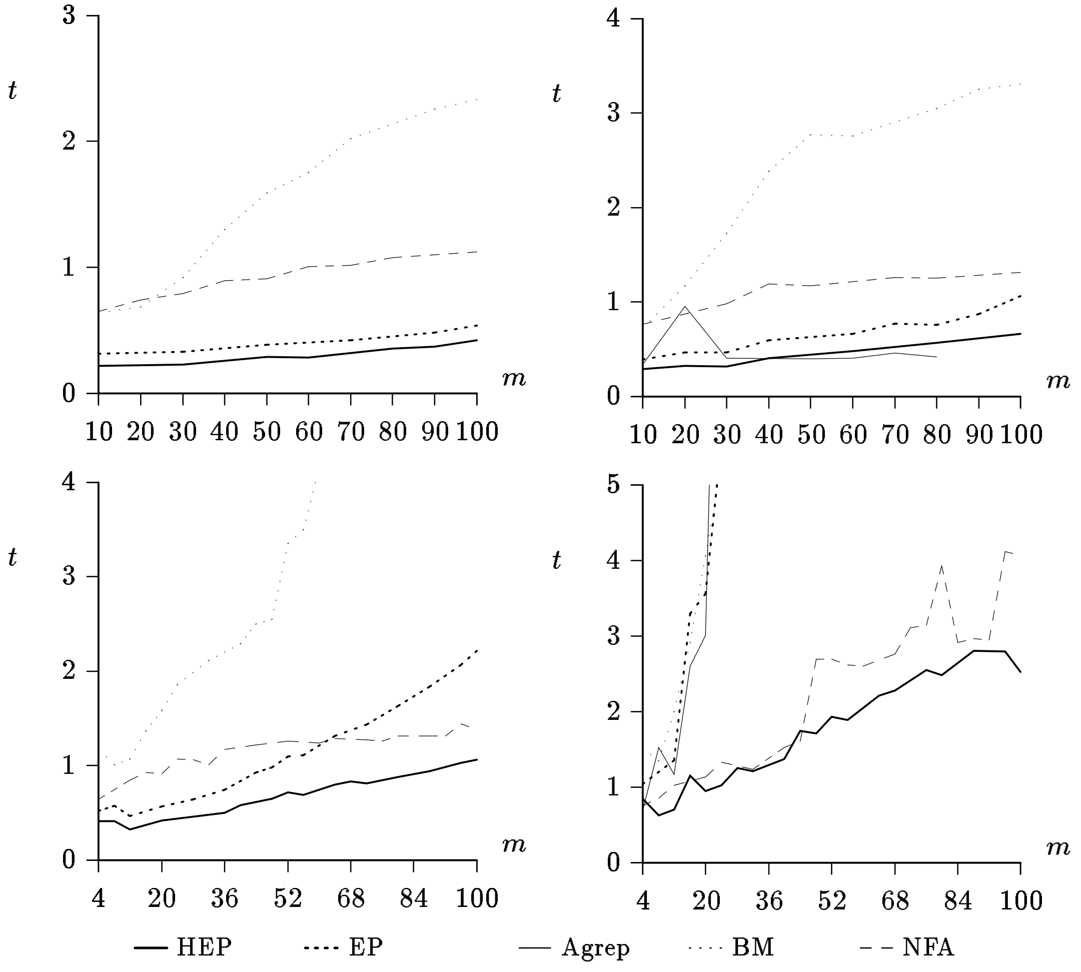


Figure 6.5: Times for long patterns. On the left, random text ($\sigma = 32$), on the right, English text. From top to bottom, the plots are for $\alpha = 0.1$ and $\alpha = 0.25$.

We are working on better cost functions for the splitting optimization technique. We also plan to study the on-line effect of splitting the pattern in more than $k+1$ pieces (so that more than one piece has to match), as suggested in [Shi96] for off-line searching.

## 6.2 A Counting Filter

We present in this section a very simple and efficient algorithm for on-line approximate string matching. It is based on a previously known counting-based filter [JTU96] that searches for a single pattern by quickly discarding uninteresting parts of the text. We give a simplified implementation of the algorithm as well as a novel analysis. We also extend the filter to improve its efficiency for low error levels, reaching $O(\alpha e^{m/\sigma}\, n)$ search cost. This filter is used later in Section 7.3 for multipattern approximate searching.

The algorithm that we extend is a filter based on counting matching positions [JTU96] [2]. The filter is linear on average, and as any filtration algorithm, is useful up to a certain $\alpha$ value. Its strongest

[2]The real story is that we reinvented (a simpler version of) this algorithm by the time in which it was accepted in Software Practice and Experience but not yet published.

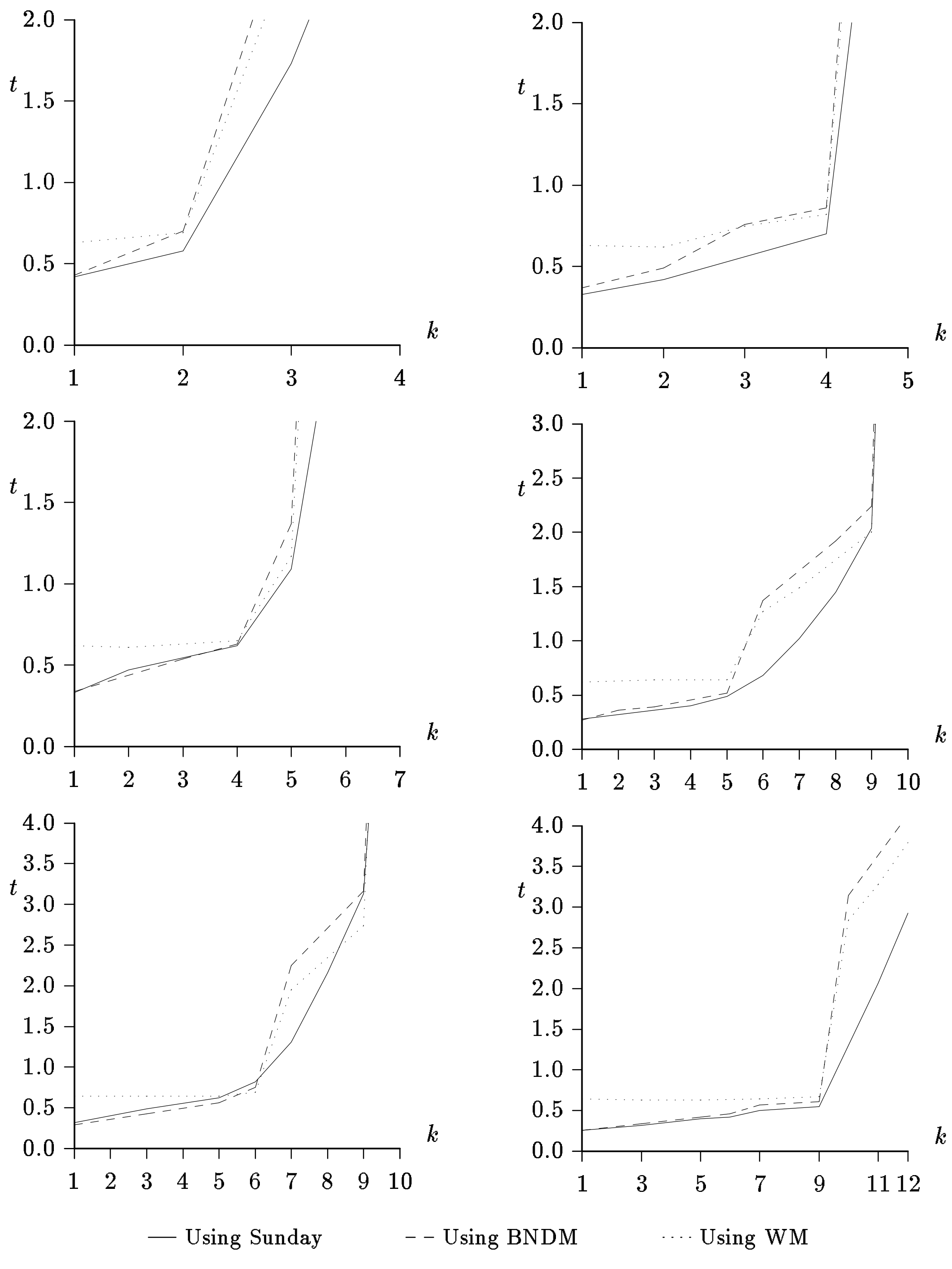


Figure 6.6: Times for random text on patterns of length 10 , 20 and 30 (one per row), and $\sigma = 16$ and 64 (first and second column, respectively).

point is its extreme simplicity. Despite that simplicity, it is among the fastest ones in its area of usefulness.

### 6.2.1 A Simple Counting Filter

In this section we describe a minor variation of [JTU96] (also very close to [GL89]). Our approach is simpler because we use a fixed-size instead of variable-size text window (a possibility already noted in [Ukk92]).

We begin by proving a very simple lemma, which is a special case ($q = 1$) of Lemma 7 of [JU91].

**Lemma:** If there are $i \leq j$ such that $ed(T_{i..j}, P) \leq k$, then $T_{j-m+1..j}$ includes at least $m - k$ characters of $P$.

**Proof:** Suppose the opposite. If $j - i < m$, then we observe that there are less than $m - k$ characters of $P$ in $T_{i..j}$. Hence, more than $k$ characters must be deleted from $P$ to match the text. If $j - i \geq m$, we observe that there are more than $k$ characters in $T_{i..j}$ that are not in $P$, and hence we must insert more than $k$ characters in $P$ to match the text. A contradiction in both cases.

Note that in case of repeated characters in the pattern, they must be counted as different occurrences. For example, if we search **aaaa** with one error in the text, the last four letters of each occurrence must include at least three **a**'s.

The filter is based on the lemma. It passes over the text examining an $m$-letters long window. It keeps track of how many characters of $P$ are present in the current text window (accounting for multiplicities too). If, at a given text position $j$, $m - k$ or more characters of $P$ are in the window $T_{j-m+1..j}$, the window area is verified with a classical algorithm (e.g. [Ukk85b]). Verification is of course necessary, since the characters of the text could be at different positions in the pattern.

To avoid re-verification due to overlapping areas, we keep track of the last position verified and the state of the verification algorithm. If a new verification requirement starts before the last verified position, we start the verification from the last verified position, avoiding to re-verify the preceding area.

Observe that it is not necessary to verify the longer area $T_{j-m-k+1..j}$ (what would be the obvious area, since the occurrence can be of length up to $m+k$). This is because the lemma holds also for the window at *any* position inside an occurrence, so that the counter will reach $m - k$ also $m$ characters past the *beginning* of the occurrence. A longer occurrence will keep triggering verifications while the window is inside the occurrence. This fact, together with our mechanism to avoid re-verifications by keeping the current state of verification, ensures that the occurrence will be caught.

We implement the filtering algorithm as follows: we build a table $A$ where, for each character $c \in \Sigma$, the number of times that $c$ appears in $P$ is initially stored. Throughout the algorithm, each entry of $A$ indicates how many occurrences of that character can still be taken as belonging to $P$. We also keep a counter *count* of matching characters. To advance the window, we must include the new character $T_{j+1}$ and exclude the last character, $T_{j-m+1}$. To include the new character, we subtract one at the proper entry of $A$. If the entry was greater than zero before the operation, it is because the character is in $P$, so we increment the counter *count*. To exclude the old character, we add one at the proper entry of $A$. If the entry is greater than zero after the operation, it is because the character was in $P$, so we decrement *count*. When the counter reaches $m - k$ we verify the preceding area.

When $A[c]$ is negative, it means that the character $c$ must leave the window $-A[c]$ times before we accept it again as belonging to the pattern. For example, if we run the pattern "aloha" over the text "aaaaaaaa", it will hold $A['a'] = -3$, and the value of *count* will be 2. Figure 6.7 illustrates.

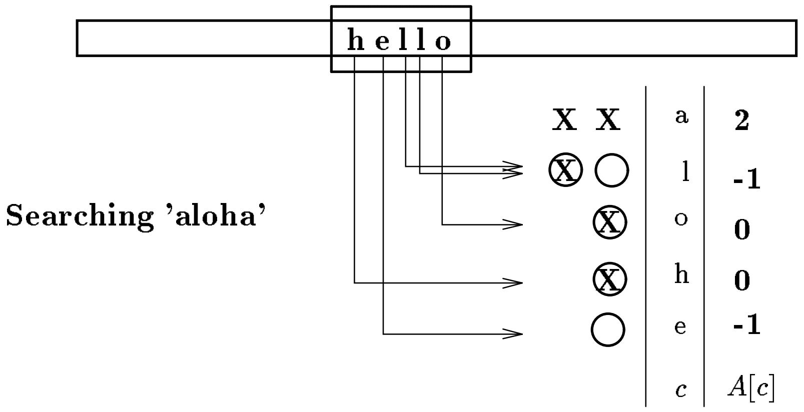


Figure 6.7: An example of the counting filter. The crosses represent elements which $A$ accepts, and the circles are the elements that appeared in the window. $A[c]$ stores circles minus crosses, and *count* counts circled crosses.

Figure 6.8 shows the pseudocode of the algorithm. As it can be seen, the algorithm is not only linear (excluding verifications), but the number of operations per character is very small.

```
    CountFilter ($T$,$n$,$P$,$m$,$k$)
{                              /* preprocessing */
       for ($c \in \Sigma$) $A[c] = 0$;
       for ($i = 1; i \le m; i{+}{+}$) $A[P_i]{+}{+}$;
       $count = -(m - k)$;
                               /* searching */
       for ($j = 1; j \le m; j{+}{+}$) /* fill the initial window */
          if ($A[T_j]{-}{-} > 0$) $count{+}{+}$;
       for ($; j \le m; j{+}{+}$) /* move the window */
        { if ($count \ge 0$) { verify $T_{j-m..j-1}$ with dynamic programming }
          if ($++A[T_{j-m}] > 0$) $count{-}{-}$;
          if ($A[T_j]{-}{-} > 0$) $count{+}{+}$;
        }
    }
```

Figure 6.8: The code of the filtering algorithm.

Finally, we notice that classes of characters can be used with this algorithm. If the pattern matches a set of characters $C_i$ at position $i$, then we simply increment $A[c]$ for all $c \in C_i$. This, however, may degrade the filtering capability of the algorithm.

### 6.2.2 Analysis

The space requirement of the algorithm is $O(\sigma)$. The preprocessing cost is $O(\sigma + m)$. If the number of verifications is negligible, the algorithm is $O(n)$ search time.

In the worst case all the text positions are verified, and the algorithm takes the same as dynamic programming, i.e. $O(mn)$. This is because we avoid re-verifying a text position, even in the case of overlapping verification requirements.

The difficult part of the analysis is the maximum error ratio $\alpha$ that the filtration scheme can tolerate while keeping the number of verifications low. If the probability of verifying is $O(1/m^2)$ the algorithm keeps linear on average. If it exceeds $1/m$, it becomes completely ineffective. This is because the verifications cost $O(m^2)$, and hence this is the point where the algorithm becomes $O(mn)$, the same as plain dynamic programming. We call that point the "limit of usability", and say that the algorithm is "useful" before that limit. We present two different analysis: first an exact analysis and second a more usable one.

#### 6.2.2.1 Exact Analysis

We obtain the probability $Pr(m, k, \sigma)$ of triggering a verification at a given text position, when the pattern and the text are random and uniformly distributed over an alphabet of size $\sigma$.

We analyze the case in which the pattern has all its letters different. We model the counting process as follows: consider a set of $\sigma$ *urns* of unlimited capacity, one per character of the alphabet. From those $\sigma$ urns, $m$ represent the characters that belong to the pattern. We say that those urns "belong" to the pattern. We scan the $m$ characters of the text window, and put each one into its corresponding urn. What the counter of our algorithm keeps is thus the number of nonempty urns belonging to the pattern. When that number of urns reaches $m - k$ a verification is triggered.

Hence, we want to compute first the probability of, given $m$ already selected urns out of $\sigma$, randomly throwing $m$ balls and at the end having *exactly* $j$ empty urns from the selected ones.

We use exponential generating functions (*egf*) and the symbolic method [SF96]. Since the text characters put in the urns are distinguishable (because different orderings produce different texts) we use labeled objects. The ordering inside a urn does not count, hence there is one urn of each size and its *egf* is $e^z$. A nonempty urn cannot have size zero, hence its *egf* is $e^z - 1$. Finally, we can select the $j$ empty urns out of the $m$ distinguished ones. The *egf* of the total number of text windows leaving exactly $j$ empty distinguished urns, $m - j$ nonempty distinguished urns and with no restriction on the remaining $\sigma - m$ urns is

$$p_{m,j,\sigma}(z) = \binom{m}{j} \left(e^z - 1\right)^{m-j} e^{z(\sigma - m)}$$

where the variable $z$ counts the number of balls used, i.e. the size of the text window. Therefore, we want the coefficient of $z^m/m!$.

The part $(e^z - 1)^{m-j}$ can be written as [AS72, page 824]

$$(m-j)! \sum_{n \geq m-j} \left\{ \begin{matrix} n \\ m-j \end{matrix} \right\} \frac{z^n}{n!}$$

where $\left\{ {n \atop m} \right\}$ is the Stirling number of the second kind (the number of ways to partition a set of $n$ elements into $m$ nonempty subsets).

We expand the expression of $p_{m,j,\sigma}(z)$ to obtain

$$\binom{m}{j} \left( (m-j)! \sum_{m_1 \geq m-j} \left\{ \begin{matrix} m_1 \\ m-j \end{matrix} \right\} \frac{z^{m_1}}{m_1!} \right) \left( \sum_{m_2 \geq 0} \frac{(\sigma - m)^{m_2} z^{m_2}}{m_2!} \right)$$

from where the coefficient of $z^m/m!$ can be extracted. This is

$$\frac{m!}{j!}\sum_{m_1+m_2=m}\left\{\begin{matrix} m_1 \\ m-j \end{matrix}\right\}(\sigma-m)^{m_2}\binom{m}{m_1}$$

by renaming $r = m_1$ and $m_2 = m - r$ we get

$$\frac{m!}{j!}\sum_{r=m-j}^{m}\binom{m}{r}\left\{\begin{matrix} r \\ m-j \end{matrix}\right\}(\sigma-m)^{m-r}$$

which by expanding the Stirling number and dividing by the total number of possible text windows configurations becomes

$$p(m,j,\sigma) = \frac{1}{\sigma^m}\binom{m}{j}\sum_{i=0}^{m-j}\binom{m-j}{i}(-1)^{m-j-i}(\sigma-m+i)^m$$

(which is easier to compute).

The same result can of course be obtained without generating functions, where $\binom{m}{j}$ corresponds to selecting the $j$ empty urns, $r$ representing the number of characters of the text window that fall into the distinguished urns, $(m-j)!\{\begin{smallmatrix} r \\ m-j \end{smallmatrix}\}$ the ways to fill the nonempty urns and $(\sigma-m)^{m-r}$ the ways to distribute the rest of the characters in the unrestricted urns.

Since the probability of triggering a verification is equivalent to having exactly $j$ empty urns, for disjoint cases $j = 0..k$, we get

$$Pr(m,k,\sigma) = \sum_{j=0}^{k} p(m,j,\sigma)$$

If the pattern has repeated characters, the probability of triggering a verification is smaller. This is easily seen by imagining that we make a given character of the pattern equal to another one. This makes a number of verification-triggering text windows that were different because of ordering these two characters differently to be equal now. However, the exact analysis is very difficult.

Our formula is very accurate but gives little intuition about its meaning. We derive now a pessimistic bound for the limit of linearity and usability.

#### 6.2.2.2 A Simpler Formula

We find an upper bound for the probability of triggering a verification, and use it to derive a safe limit for $\alpha$ to make verification costs negligible. We consider constant $\alpha$ and varying $m$ (the results are therefore a limit on $\alpha$). We then extend the results to the other cases.

The upper bound is obtained by using a pessimistic model which is simpler than reality. We assume that every time a letter in the text window matches the pattern, it is counted regardless of how many times it appeared in the window. Therefore, if we search `aloha` with 1 error in the text window `aaaaa` the verification will be triggered because there are 5 letters in the pattern (where in fact our counter will not trigger a verification because it counts only 2 `a`'s).

Consider a given letter in the text window. The probability of that letter being counted is that of appearing in the pattern. This is the same as being equal to some letter of the pattern. The probability of *not* being equal to a given letter is $(1 - 1/\sigma)$. The probability of not being in the pattern is therefore $p = (1 - 1/\sigma)^m$.

In our simplified model, each pattern letter is counted independently of the rest. Therefore the number $X$ of letters in the text window that did not match the pattern is the sum of $m$ (window length) random variables that take the value 1 with probability $p$, and zero otherwise. This has a Binomial distribution $B(m, p)$.

Therefore, our question is when the probability $Pr(X \leq k)$ is $O(1/m^2)$ (so that the algorithm is linear) or when it is $O(1/m)$ (so that it is useful). In the proof we use $O(1/m^2)$, since as we will see shortly the result is the same for any polynomial in $1/m$.

We first analyze the case where the mean of the distribution is beyond $k$, i.e. $mp > k$. This is the same as the condition $\alpha < p$. As $Pr(X = j)$ increases with $j$ for $j < mp$, we have $Pr(X \leq k) \leq k\ Pr(X = k)$.

Therefore, it suffices to prove that $Pr(X = k) = O(1/m^3)$ for linearity or that $Pr(X = k) = O(1/m^2)$ for usefulness. By using the Stirling approximation to the factorial we have

$$Pr(X = k) \quad = \quad \binom{m}{k} p^k (1-p)^{m-k} \quad = \quad \frac{m^m p^k (1-p)^{m-k}}{k^k (m-k)^{m-k}}\ O(\sqrt{m})$$

which can be rewritten as

$$\left( \frac{p^\alpha (1-p)^{1-\alpha}}{\alpha^\alpha (1-\alpha)^{1-\alpha}} \right)^m\ O(\sqrt{m})$$

It is clear that the above formula is $O(1/m)$ or $O(1/m^2)$ or $O(1/m^3)$ whenever the base of the exponential is $< 1$. This is

$$p^\alpha (1-p)^{1-\alpha} < \alpha^\alpha (1-\alpha)^{1-\alpha} \tag{6.1}$$

To determine the cases where the above condition is valid, we define the function

$$f(x) = x^\alpha (1-x)^{1-\alpha}$$

which reaches its maximum at $x = \alpha$. This shows that Eq. (6.1) holds everywhere, and therefore the probability of matching is $O(1/m^2)$ in the area under consideration, i.e. whenever $\alpha < p$.

On the other hand, if the mean of the distribution is less than $k$, then just the term of the summation corresponding to the mean $r = mp$ is (using Stirling again)

$$\binom{m}{mp} p^{mp} (1-p)^{m(1-p)} \quad = \quad \left( \frac{p^p (1-p)^{1-p}}{p^p (1-p)^{1-p}} \right)^m\ \Omega(m^{-1/2}) \quad = \quad \Omega(m^{-1/2}) \tag{6.2}$$

which is not $O(1/m)$.

Therefore, we arrive at the conclusion that the filter is linear and useful whenever

$$\alpha < p = \left(1 - \frac{1}{\sigma}\right)^m = e^{-m/\sigma}\ \ (1 + O(1/\sigma)) \tag{6.3}$$

and is not useful otherwise.

We have considered the case of constant $\alpha = k/m$. Obviously, the filter is linear for $k = o(m)$ and is not useful for $k = m - o(m)$. The unexplored area is $k = mp - o(m)$. It is easy to see that the filter is not useful in this case, by considering $Pr(X = mp - \epsilon)$ with $\epsilon = o(m)$, and using Stirling as in Eq. (6.2). The resulting condition is $1 - \epsilon^2/(m^2 p(1-p)) = O(m^{-1/2})$, which does not hold for any $\epsilon = o(m)$.

### 6.2.3 A Sampling Technique

One of the drawbacks of the filter is its inability to skip characters, that is, it has to inspect every text character. This puts it in disadvantage against other sublinear filters like that of Section 6.1. In this section we show how we can skip characters.

To imagine this, it is better to think that the filter counts the number of mismatches, and it triggers a verification when it cannot find more than $k$ mismatches. Imagine now that we examine one out of $s$ text characters. To avoid loosing a match, we must pessimistically consider that the character which has not been examined is present in the pattern. That is, we examine only some of the window positions and we must obtain more than $k$ mismatches from the examined positions only. It is clear that, the larger $s$, the faster the algorithm but the less tolerant to the error level it is, i.e. it will be harder to find the $k$ mismatches and to avoid the verification of the window.

To analyze this algorithm, consider that we have now $m' = m/s$ window positions to reject the pattern (i.e. to collect more than $k$ characters not present in the pattern) but the number $k$ does not change. However, the pattern has still $m$ letters. The analysis of Section 6.2.2.2 can be reused by noticing that the pattern length is used to compute $p$, while the rest of the analysis uses $m$ as the number of window positions inspected. Hence, if we inspect one position out of $s$, the maximum tolerable error level is

$$k/m' < e^{-m/\sigma} \; (1 + O(1/\sigma))$$

or equivalently

$$s < e^{-m/\sigma}/\alpha$$

Finally, our search cost is $O(n/s)$, i.e.

$$O\left(\alpha e^{m/\sigma} \; n\right)$$

### 6.2.4 Experiments

We first show experiments about the maximum allowable error ratio for the filter (i.e. up to where it is better than plain dynamic programming). Later, we compare our algorithms against others.

#### 6.2.4.1 Maximum Error Ratio

We experimentally find out which is the limit of usability of the algorithm for different types of texts, and use least squares to find a formula which is very accurate for the range of values we are interested in practice, i.e. $m \leq 100$ and $20 \leq \sigma \leq 60$. That type of formula was selected among a number of classes we tried, since it gave us the best results. It is close in spirit to Eq. (6.3) (recall that that equation is pessimistic).

The experiments were carried out as follows. For every $\sigma$ in the set $\{20, 30...60\}$ and every $m$ in $\{4..100\}$, we generated a random text of 1 Mb, and repeated 100 times the experiment of generating a random pattern and verifying which was the maximum error ($k$) up to where the number of verifications triggered was less than $1/m$ times the size of the text.

Separately for each value of $\sigma$, we used least squares for the model $\alpha_{\max} = ab^m$, which gave us the best results. Later, once a different value of $a$ and $b$ was obtained for each $\sigma$, we used the models $a = c\sigma^d$ and $b = 1 - f\sigma^g$. The result is the formula

$$\alpha_{\max} = 0.11 \; \sigma^{0.43}(1 - 0.032/\sigma^{0.37})^m$$

for which we obtained an average squared error near 0.0004 (its square root being 0.02).

We also performed the test on English text. The experimental results are shown in Figure 6.9. The jumps in the experimental curves are not due to variance in the experiments, but to the complexity of the combinatorial nature of the process (the same jumps appear in the resulting formula of Section 6.2.2.1). The smooth curves are those obtained with least squares. Our theoretical pessimistic approximations are totally below the experimental curves, but have the same shape of those of least squares. Therefore, they are less exact for very small or very large $m$. In the first case this is because the analysis works with probabilities of the form $O(1/m)$, which allows larger errors for small $m$. In the second case it is because the pessimistic part of the model refers to letters that appear many times in the text window of length $m$, which is more noticeable for large $m$ (when it is more probable to repeat letters).

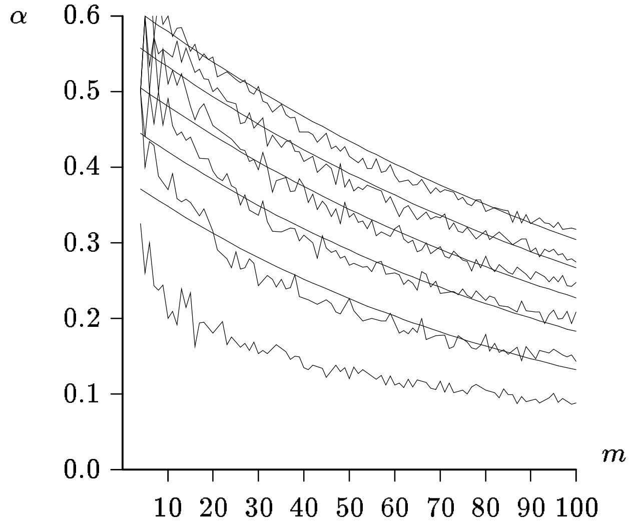


Figure 6.9: Experimental maximum level of usefulness of our algorithm. The lowest line is for English text. The rest of non-smooth lines are for $\sigma = 20, 30...60$ (from lower to upper). The smooth lines are our approximation.

#### 6.2.4.2 Comparison among Algorithms

In this section we experimentally compare the different versions of our algorithms among them and against EP (see Section 5.9). Figure 6.10 compares our basic algorithm and the sampling technique, as well as EP. It can be seen that sparser sampling yields a faster algorithm if the error level is low enough, otherwise it is counterproductive (in particular, the sampling idea is of useless on English text). As it can be seen, however, EP is faster when it works well.

To compare our algorithm against the others, we refer the reader to Figures 5.16 and 5.17 (our implementation is called "Count"). As it can be seen, this algorithm is not the fastest, but it is very simple and competitive. The algorithms which improve over it come all from this same thesis, except for Myers' BPM algorithm [Mye98] which appeared later. At the moment it was published, the algorithm was the fastest in a small $\alpha$ band, since although not as fast as sublinear filters like EP, it was more resistant to the error level (our improvement of Section 6.1 did not exist by then).

Apart from the simplicity of the algorithm, the value of this section is in the novel analysis of the area of applicability and in that the algorithm is the basis for the development of a multipattern

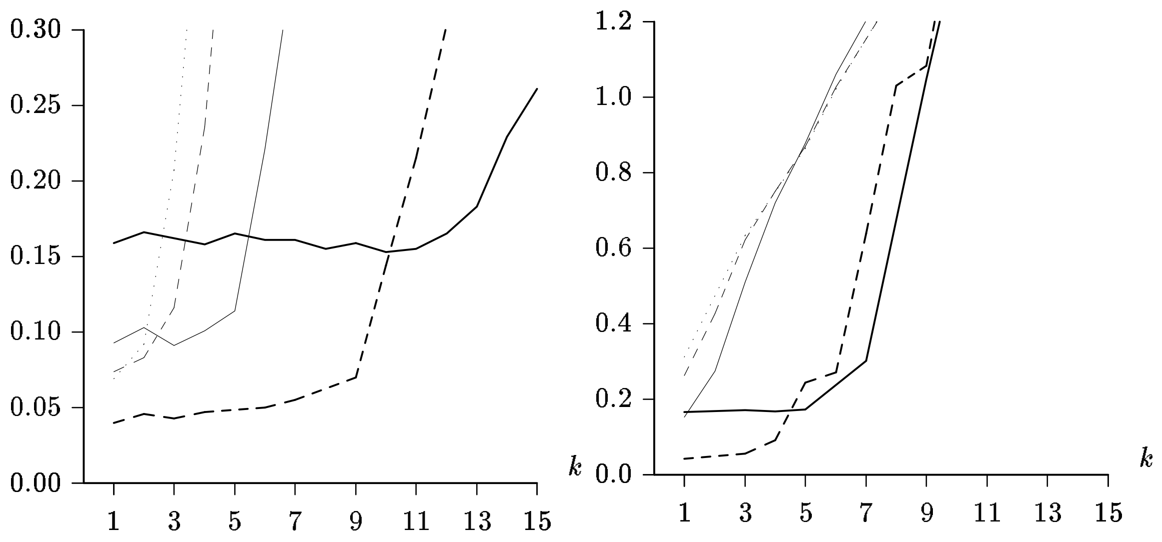


Figure 6.10: Our algorithm using different steps versus EP (thick dashed line). The thick solid line is the original algorithm and the thin lines show the use of the sampling technique taking one sample each 2,3 and 4 characters (solid, dashed and dotted lines, respectively). We show megabytes per second for $m = 30$, for random (left, $\sigma = 64$) and English text (right).

search algorithm which is the fastest one for intermediate error levels (Section 7.3).

Extending this filter to different distance functions is not difficult, it is just a matter of determining how many letters must be present in an approximate occurrence. Instead of $m - k$, we must use this number. Allowing transposition is especially easy since we do not account for the positions of the letters in the filter. If only transpositions were allowed, we would use this filter allowing zero errors.

## 6.3 A Suffix Automaton Approach

We present a new filtering algorithm based on the combination of our NFA and a suffix automaton. The NFA is modified so that it recognizes every suffix of the pattern allowing $k$ errors. The result is an algorithm which is able to skip characters and to handle at the same time pattern extensions such as classes of characters. A lower bound on its average complexity is $O(\alpha n/(1-\alpha))$, and therefore it is not useful for $\alpha > 1/2$. Experimental results show that it is competitive against the fastest filtration algorithms, and that in some restricted cases it is the fastest known filter. This idea is included in [NR98b].

### 6.3.1 Adapting the NFA

Section 2.4 presented a nondeterministic finite automaton (NFA) which is built from the pattern and the number of errors and recognizes all the approximate occurrences of the pattern in a text. This automaton is simulated using the bits of the computer word: row-wise (i.e. packing the rows in computer words) in [WM92a] and diagonal-wise in Chapter 5. As we have shown, bit-parallelism has the advantage that it allows performing more flexible searching.

On the other hand, we have shown in Section 2.8 how bit-parallelism can be used to simulate a

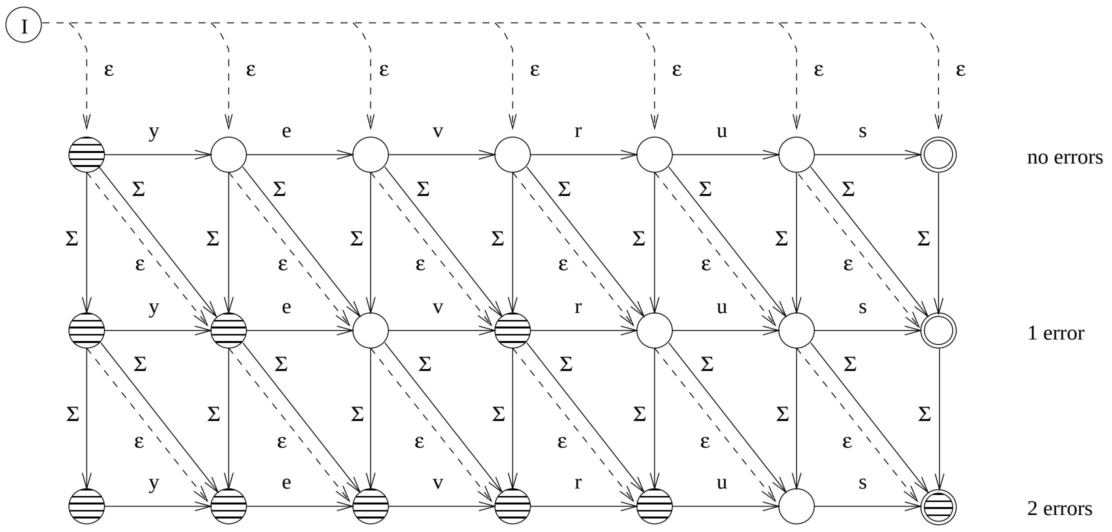


Figure 6.11: Our NFA to recognize suffixes of the pattern "`survey`" reversed.

*suffix automaton*, which is used for exact string matching (algorithm BNDM, which is an extension of BDM [CCG+94, CR94]).

We now merge both approaches. We modify the NFA so that it recognizes not only the whole pattern but also any suffix of the pattern, allowing up to $k$ errors. We also modify it so that it computes edit distance from the point it is started. As for BNDM, we build the automaton on the reversed pattern. Figure 6.11 illustrates the modified NFA.

Consider the initial state "I" we added. The $\varepsilon$-transitions leaving from the initial state allow the automaton to recognize with $k$ errors not only the whole pattern but also any suffix of it. Our second modification on the original automaton of Section 2.4 is the removal of the self-loop at the top-left state, which allowed it to start a match at any text position. Our automaton, therefore, recognizes suffixes of the pattern which start at the beginning of the text window.

### 6.3.2 The Search Algorithm

We move a window over the text, and we are interested only in occurrences that start at the current window position. Any occurrence has a length between $m-k$ and $m+k$. If there is an occurrence of the pattern $P$ starting at the window position with $k$ errors, then a prefix of $P$ must match the first $m-k$ characters with $k$ errors. Hence, we cannot miss an occurrence if we keep count of the matches of all the pattern prefixes in a window of length $m-k$. If there are no more pattern substrings matching with $k$ errors, then we cannot miss an occurrence and we can shift the window to the last prefix that matched (with errors).

To keep count of the pattern prefixes that match with errors, we use the adapted automaton. The search process inherits from BNDM, as follows. We move a window of length $m-k$ on the text, and search backwards a suffix of the window which matches the pattern with at most $k$ errors. This search is done using the modified NFA explained above, which is built on the reversed pattern. We remember in the variable *last* the longest suffix of the window that matches a prefix of the pattern (in fact, a prefix of the reversed pattern read backwards) with a distance less or equal to $k$. This is done in constant time by checking whether the rightmost bottom state of the NFA is active. On the other hand, if the NFA runs out of active states we know that a match is not possible in the window (since no pattern substring matches a suffix of the window with $k$ errors or less) and we can shift to the *last* position where we found a pattern prefix, as in the exact matching algorithm. Notice that the automaton has no active states if and only if its last row has no active states.

Each time we move the window to a new position we restart the automaton with all its states active, which represents setting the initial state to active and letting the $\varepsilon$-transitions flush this activation to all the automaton (the states in the lower left triangle are also activated to allow the deletion of the first letters of the pattern). If after reading the whole window the automaton still has active states, then it is possible that the current window starts an occurrence, so we use the traditional automaton to compute the edit distance from the initial window position in the text. After reading at most $m + k$ characters we have either found a match starting at the window position or left the automaton without active states.

Notice that if the automaton has active states after reading the complete window, then a match starting at the window is *possible* and we have to check it explicitly since we can only ensure that a substring of the pattern matches in the window.

The automaton can be simulated in a number of ways. Wu and Manber do it row-wise (each row of the automaton is packed in a computer word), while in Chapter 5 we do it diagonal-wise. In this case we prefer the technique of Wu and Manber, since in our approach the initial diagonals of length $\leq k$ are discarded, and they are needed here. Although we can adapt our automaton to compute edit distance, this will require more computer words, which will be all active because we start with all 1's. On the other hand, this approach is good only for very small $k$ values and intermediate $m$, where Wu and Manber use $k+1$ computer words and our approach needs the same number or even more words. As shown in Chapter 5, the speed of the row-wise implementation is similar to that of the diagonal-wise for very small $k$ if we put the computer words in registers.

### 6.3.3 Analysis

As the automaton cannot run out of active states before examining more than $k$ letters, and we skip at most $m - k$ positions, a lower bound on the complexity of this algorithm is $\Omega(kn/(m-k))$. We show now that the average case is not very different.

We know from Section 4.1 that the probability of matching has a very abrupt nature, jumping from almost zero to almost one in a short period. Specifically, if $\alpha < 1 - e/\sqrt{\sigma}$ the probability is exponentially decreasing on $m$, and it becomes at least $\Omega(1/m)$ after that point.

When we are traversing the window backwards, the automaton is alive after reading $i$ letters whenever some substring of length $i$ of the pattern matches the text with $k$ errors or less. As there are $m-i \leq m$ such substrings, we have that the probability of some substring matching is exponentially decreasing with $i$ whenever $k/i < 1 - e/\sqrt{\sigma}$, and becomes almost 1 for smaller $i$. Hence, on average we inspect $k/(1 - e/\sqrt{\sigma})$ letters with very high probability, and at the next letter the automaton normally has no more active states.

On the other hand, when we shift the window we align it to the last time where the automaton had a final state active. Since this phenomenon is so abrupt, the probability of a suffix matching is similar (except for polynomial factors in $i$) to that of a substring matching, so on average we shift exactly to the position following the place where the automaton was without active states.

Hence, we work on $k/(1-e/\sqrt{\sigma})$ letters and then shift $(m-k) - k/(1-e/\sqrt{\sigma})$ positions. Therefore, our average complexity is

$$O\left(\frac{\alpha n}{(1-\alpha)(1-e/\sqrt{\sigma})-\alpha}\right)$$

where we have excluded the cost of verifications, which is assumed to be negligible since we suppose $\alpha < 1 - e/\sqrt{\sigma}$. The above formula is sublinear for

$$\alpha \quad < \quad \frac{1-e/\sqrt{\sigma}}{3-e/\sqrt{\sigma}}$$

which for large alphabets tends to 1/3.

If we have a long pattern we have to multiply the above formula by $O(m/w)$, which is the per-character cost of a multi-word simulation.

### 6.3.4 Experimental Results

We compare now the performance of our algorithm against others. The codes are the same as those of Section 5.9, and we have selected only the fastest ones for this case. Figure 6.12 shows the results. As the algorithm works well for very low error levels, we show only the cases $k = 1$ to 3, for random ($\sigma = 4$) and English text. In the first case our algorithm outperforms all the others (including that of Section 6.1, which is in general the fastest one). For English text it does not, although it is competitive for $k = 1$ and intermediate pattern lengths. This is also the case of other alphabet sizes. Notice that we have implemented specialized code for each fixed $k$ value, and that a general implementation would probably be much slower. Hence, this algorithm should be regarded as a good choice for very low error levels.

More refinements are possible for this algorithm. For instance, we can avoid verifying a text window if it does not start with a match. This is achieved with the same mechanism used in Chapter 5 to eliminate the states of the upper-right triangle of shorter diagonals and clear the last complete diagonal at each match. Although, we have obtained appreciable speedups applying these techniques, these occur for high error levels where the algorithm is not competitive anyway, while the performance for low error levels (where the algorithm is competitive) degraded.

Notice that the only algorithm that beats significantly this one is that of Section 6.1, i.e. also part of this thesis.

## 6.4 A Partial Deterministic Automaton

One of the simplest approaches to approximate string matching is to consider the associated non-deterministic finite automaton and make it deterministic. Besides automaton generation, the search time is $O(n)$ in the worst case. This solution is mentioned in the classical literature but has not been pursued further, due to the large number of automaton states that may be generated.

We study the idea of generating the deterministic automaton on the fly. That is, we only generate the states that are actually reached when the text is traversed. We show that this limits drastically the number of states actually generated. Moreover, the algorithm is quite competitive. We present some empirical results on the growth of the automaton.

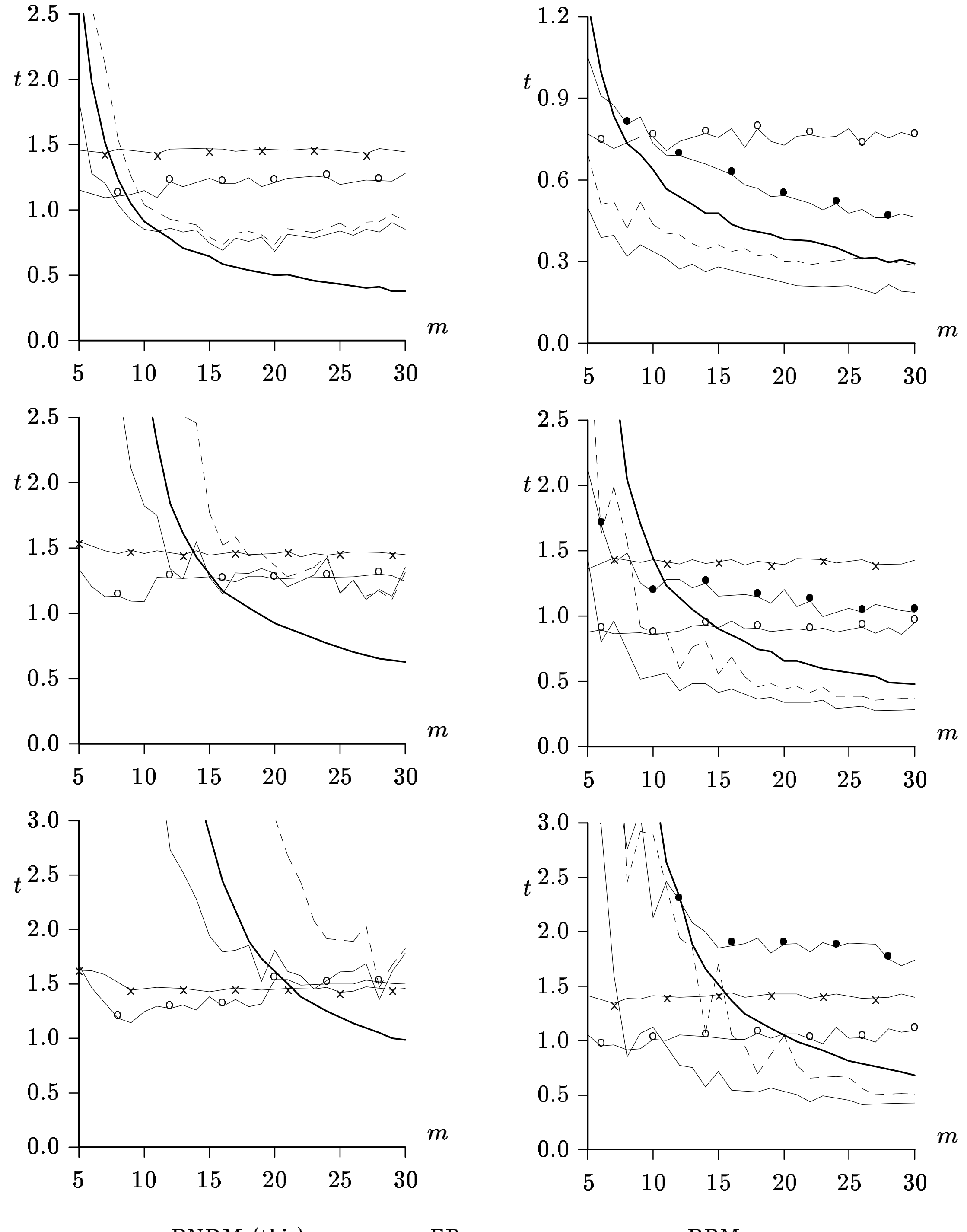


Figure 6.12: Times in 1/10-th of seconds per megabyte, for random text ($\sigma = 4$, on the left) and English text (on the right), $k = 1$ to 3 (first, second and third row, respectively). The $x$ axis is the pattern length.

### 6.4.1 Lazy Automata

As explained in Chapter 2, the problem of approximate string matching can be solved using a matrix or using a nondeterministic finite automaton (NFA). In any case we have a concept of *state of the search*, which is given in the first case by the set of values of the current matrix column and by the set of active states in the second case. In [Ukk85b] the use of a deterministic finite automaton (DFA) is proposed, where each "state of the search" is converted into a state of the automaton. Each state could be identified, for instance, with the set of values of the current column of the dynamic programming matrix. Building this automaton is equivalent to precompute all possible transitions between different columns instead of doing it on the fly as the text is processed. Once the automaton is built we need only $O(n)$ time to process the text instead of $O(mn)$, because the transition between columns is done at $O(1)$ cost instead of $O(m)$ as in classical dynamic programming.

To reduce the number of states, the property of "active columns" is used in [Ukk85b] (see Section 3.1.1), so that every column value larger than $k+1$ is converted to $k+1$, this way reducing the number of states without affecting the output of the algorithm. Unfortunately, as shown in Section 3.1.2, the number of states of this automaton is very large, which means that the preprocessing time and space requirement is not acceptable in practice.

We return to the idea of building the DFA. However, we observe that most of the states of the DFA are never reached throughout the search. Therefore, instead of building the automaton beforehand and then using it to search the text, we have a partially built automaton. This partial automaton has only the states and transitions that have been reached. As we search, if we find a transition which has not been computed yet, we compute it before proceeding. This is as efficient as building the automaton completely, but the number of states can be much smaller.

The idea of on-the-fly construction of DFAs is not new. For instance, it has been mentioned before, although not exploited, for the more restricted case of string matching allowing character replacements (no deletions nor insertions) [BYG94]. It has also been used for the general problem in [Kur96], where it was implemented on a lazy functional language. Kurtz arrives at similar conclusions about performance, although we study the idea more in depth, include more algorithms in the comparison and our test suite is much larger and has no hidden performance factors that could be included in a functional language implementation. We also study the growth of the automaton, as well as techniques to work with a limited amount of memory.

### 6.4.2 The Algorithm

We begin with some terminology. A *deterministic finite automaton* or *DFA* is a set of *states* connected by *transitions*. Transitions are arrows among states labeled with symbols drawn from an alphabet $\Sigma$. There is exactly one transition leaving every state for each alphabet symbol. One of the states is *initial* and some states are *final*. The automaton is run over a text beginning in its initial state. Given each text character, it follows the appropriate transition and reaches a new state. We say that the automaton *accepts* a text position whenever it is at a final state just after that position is read.

A *partial* DFA is a DFA where some transitions are missing. A missing transition means that we still have not computed to which state it should go.

A *configuration* represents the state of the search at a given moment. It can be represented as the set of active or inactive states of the NFA, or as the current (active) values of the the dynamic programming algorithm. Each possible configuration corresponds to a state of the DFA.

Our DFA will have one state per "known" configuration. Only the configurations (states) which are actually reached in the processing of the text will be present.

We begin with a partial DFA with just one state and all missing transitions. The state corresponds to the initial configuration ($C_i = i$). We traverse the text exactly as if we had a complete DFA. The difference is that, whenever we must follow a missing transition, we compute it. That is, we take the current state (we store the configuration that corresponds to each state) and perform an $O(m)$ step of the classical algorithm. This gives us the configuration of a new state. We search the configuration among the known states. If it already exists, we put the previously missing transition pointing to that state. Otherwise, we must first create a new state (with all transitions missing).

The advantage of such construction is that, although the DFA of a pattern can be very large, only a small portion of the states may be actually reached along the text. Of course, the larger the text, the more states will be generated, but this larger text will compensate for the effort of generating the automaton. Note that in natural language some substrings never appear, no matter how long the text is.

The only disadvantage is that once the complete DFA is generated, the configuration to which each state corresponds needs not be stored, while in the partial DFA we need to keep those configurations all the time to be able to generate new transitions and states. This extra space turns out to be about a 25% extra per *generated* state, which is not too much, especially because many fewer states are generated in the partial DFA (as we show in the experimental section). On the other hand, the complete DFA algorithm needs to keep all configurations to generate the DFA, and only then can free their space. Therefore, at some point it demands strictly more memory than what the partial DFA algorithm demands along the whole search. Figure 6.13 sketches the algorithm.

```
Search $(T, n, P, m, k)$
    $Aut \leftarrow$ initial state (configuration $C_i = i$)
    $state \leftarrow$ initial state
    $\forall c \in \Sigma$, $transition(Aut, state, c) \leftarrow unknown$
    for $(i = 1; i \le n; i{+}{+})$
        { $nstate \leftarrow transition(Aut, state, T_i)$
          if $(nstate = unknown)$
              { $nconf \leftarrow perform\_step(conf(Aut, state), T_i)$
                $nstate \leftarrow$ state in $Aut$ corresponding to $nconf$
                if ($nstate$ not found)
                    { $nstate \leftarrow$ new state
                      $Aut \leftarrow Aut \cup \{nstate\}$
                      $\forall c \in \Sigma$, $transition(Aut, nstate, c) \leftarrow unknown$
                    }
                $transition(Aut, state, T_i) \leftarrow nstate$
              }
          $state \leftarrow nstate$
          if ($state$ is final) report match
        }
```

Figure 6.13: The partial DFA algorithm.

The $\Sigma$ in the algorithm does not stand for the complete alphabet, but only for those symbols appearing in the pattern, plus one that represents "any symbol not in the pattern". The alphabet is then mapped to the interval $[0..p]$ where $p \leq m$, and therefore creating a new state costs $O(m)$. The text characters are mapped in $O(1)$ time by using a global $O(\sigma)$ size table.

Now we point out some details of the algorithm.

We represent configurations as the current $C_i$ values of the classical dynamic programming algorithm. Therefore the configurations are $O(m)$ size. The action *perform_step* of the algorithm simply makes a step of the classical algorithm on the given configuration, and therefore it is $O(m)$ time. Since we work only on active values, our representation for a column is $[a, c_1, ..., c_a]$, where $a$ is the position of the last active value. As shown in Section 4.1, the average number of active values is $O(k)$. However, this is not true if we compute each configuration only once.

Once $nconf$ has been computed, it is necessary to know whether it corresponds to a state which is already present in the automaton. Therefore, we must search in a set of "known" configurations. This search can be done in time proportional to the length of the searched configuration (i.e. $O(m)$) in the worst case. The data structure to achieve this is a trie on the $a+1$ "digits" of the configuration.

The structure of this trie is quite particular. The root has $m-k$ sub-tries (one per possible $a$ value). The sub-trie number $i$ has height $i$. Every non-root node which is not a leaf has at most 3 children. This is because the difference between two consecutive values in the column is $-1$, 0 or 1. This fact, already noticed in [Ukk85b], allows saving a lot of space in the trie implementation. This structure is used in [Kur96].

However, we found that a simple hashing turns out to be the most efficient data structure in practice. We take the hash function over the $a+1$ "digits" of the configuration. Collisions are resolved with a linked list. If the table size and the hash function are well chosen, the average cost is $O(m)$ (to evaluate the hash function), at much less space consumption.

### 6.4.3 Analysis

We call $s$ and $t$ the total number of states and transitions, respectively, in the complete automaton. This is an upper bound to those actually generated ($s'$ and $t'$, respectively). As explained in Section 3.1.2, the number of states can be upper bounded by $s = O(\min(3^m, (2\min(\sigma, m)m)^k m, (k+2)^{m-k}(k+1)!))$, where we replaced $\sigma$ by $min(\sigma, m)$ because of our character mapping (this is also noted in [Mel96]).

The number of transitions is therefore $t = O(s \min(m, \sigma))$. We have $s' = O(\min(s, n))$ and $t' = O(\min(t, n))$, since each text character can create at most one state and transition. The space needed by our algorithm is $O(s' \min(m, \sigma) + t') = O(s' \min(m, \sigma))$ in the worst case, i.e. $O(\min(s, n) \min(m, \sigma))$. The algorithm uses linear time except for the generation of the states and new transitions, each one costing $O(m)$. This makes the total cost of the algorithm $O(n + s' \min(m, \sigma) + t'm) = O(n + t'm) = O(n + m \min(t, n))$.

It is extremely difficult to set up a correct probabilistic model to compute the number of states that are generated on average after reading $n$ random symbols. We make the simplifying assumption that each new character produces a random transition. This assumption is pessimistic since some transitions are much more probable than others. The uniform model maximizes the total number of states visited.

The probability of a given transition *not* being generated at a particular text position is $(1 - 1/t)$.

Therefore, the average number of generated states after reading $n$ text characters is

$$t' = t\left(1 - \left(1 - \frac{1}{t}\right)^n\right) = t\left(1 - e^{-n/t}\right) + O(1/t)$$

and therefore the average time complexity is $O(n + mt(1 - e^{-n/t}))$. Notice that $(1 - e^{-n/t})$ is the factor by which building our partial DFA is more efficient than building the full DFA. We have three cases now

$[n = o(t)]$ In this case $e^{-n/t} = 1 - n/t + O((n/t)^2)$, which makes the total time cost $O(n + mn(1 + o(1))) = O(mn)$. That is as bad as plain dynamic programming. However, in this case the partial DFA is asymptotically better than the full DFA (i.e. partial/full $= 1 - e^{-n/t} = o(1)$).

$[n = \omega(t)]$ In this case we have that the time is $O(n + mt) = o(mn)$, and therefore we improve dynamic programming. If $n = \Omega(mt)$, then the algorithm is $O(n)$.

$[n = \Theta(t)]$ We have that $(1 - e^{-n/t}) = \Theta(1)$ (i.e. constant) and therefore the total time is $\Theta(mn)$, although the constant is smaller. For instance, if $n = t$ it is $\approx 0.63mn$.

This shows that the algorithm is competitive when the text is large compared to the size of the automaton, which is intuitively clear. However, this analysis is pessimistic and the results are much better in practice, as well as the efficiency ratio among partial and full DFA. We show this in the next section.

### 6.4.4 Experiments

We present some experimental results collected on English text. We first show the growth of the automata and then we compare our algorithm against the others. We used 10 Mb of filtered text. Each data point was obtained by averaging over 10 trials.

#### 6.4.4.1 Automaton Growth

We experimentally show in Figure 6.14 the growth of the partial and complete DFAs as the error level increases and as the traversed text increases.

As it can be seen, the size of the automaton grows slowly with the text size after a sharp start. Even after processing 10 Mb of natural language, the sizes are less than 20% of the complete automata (except for very small $k$ where the complete automata are very small anyway). Memory limitations prevented us to compute automata with more than 500,000 states (although we used a 128 Mb machine for this task). For $m = 20$ we could not generate the complete automaton past $k = 6$ and for $m = 30$ past $k = 5$. For those larger patterns our improvement is more dramatic, since the partial DFAs are completely manageable up to $k = 10$ or 11, being hundreds of times smaller. For $m = 30$ we could not even compute the partial automata past $k = 18$.

We apply now least squares over the curves to understand how the automata grow. As a function of $n$, the growth of the partial automata shows a clear $O(n^b)$ slope. Curve fitting using the model $s = an^b$ gives a relative error close to 0.25%. We show in Figure 6.15 the $a$ and $b$ values for $m = 10$, 20 and 30, as a function of the error level $\alpha$. As it can be seen, up to some point the $a$ value grows slowly and the exponent $b$ shows a somewhat linear increase. After that point, $a$ grows abruptly and the exponent stops growing (in fact, the phenomenon is so abrupt that we could not obtain the

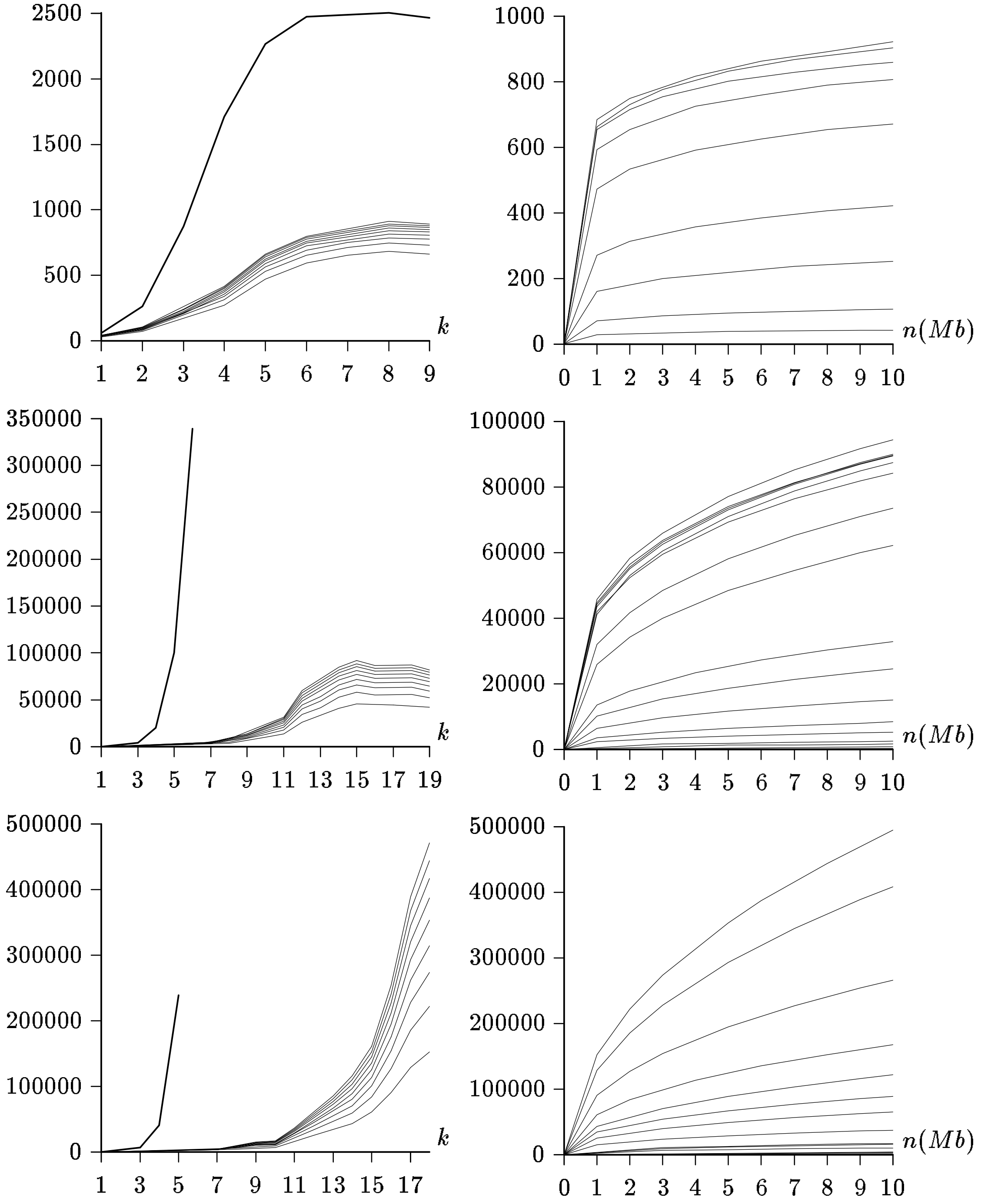


Figure 6.14: Number of states of the partial and complete DFAs. The rows are for $m = 10$, 20 and 30, respectively. The left column shows the growth in terms of $k$ (for $n$=1 to 10 Mb, from lower to upper, and the thick line for the full automaton). The right column shows the growth in terms of $n$ (for $k = 1$ to $m - 1$, in general from lower to upper).

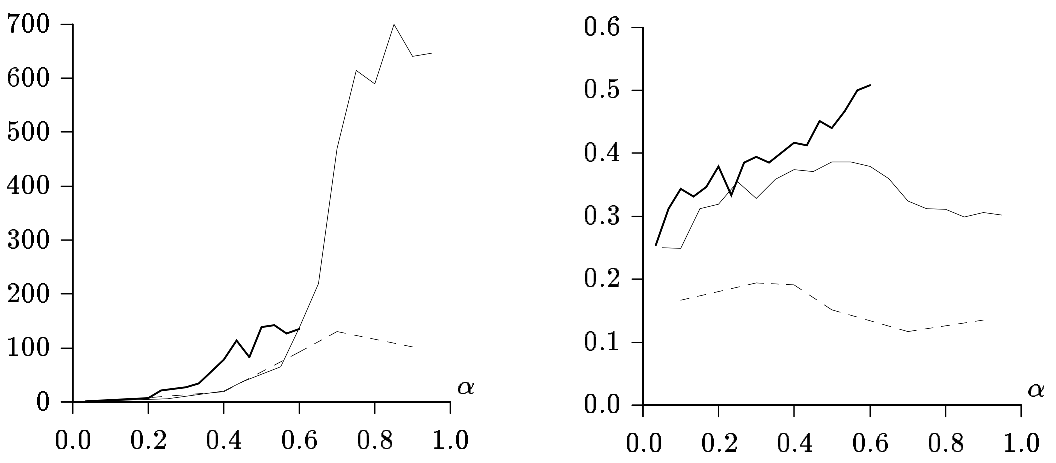


Figure 6.15: The $a$ (left) and $b$ (right) values from the fitting $s = an^b$ of the partial DFA size as a function of the text size. Dashed lines correspond to $m = 20$, solid thin lines to $m = 20$ and thick lines to $m = 30$.

values for $m = 30$ after that point). Interestingly enough, this point where the behavior changes corresponds to our limit $\alpha < 1 - 1.09/\sqrt{\sigma}$ of Eq. (4.2). Both $a$ and $b$, on the other hand, seem to increase as $m$ grows. This is not exactly true for $m = 10$, but in this case the distortion may correspond to the fact that the total number of states is not so high. If, on the other hand, we consider the growth of the automata as a function of $\alpha$, we observe the same abrupt phenomenon. Finally, we have not considered the effect of the alphabet size. A more complete study of the growth of the DFA is left for future work.

#### 6.4.4.2 Comparison Against Other Algorithms

We refer the reader to Section 5.9 for an experimental comparison between the DFA and others. For short patterns it is quite close to the fastest bit-parallel algorithms. By the time it was published, it was the fastest for intermediate error levels, i.e. between the point where filtration algorithms stopped working and where it had so many states that was slow. Later improvements (all part of this same thesis, except Myers' BPM algorithm [Mye98]) covered the areas where this algorithm was the best.

### 6.4.5 Working with Limited Memory

A problem of the automaton approach is that for long patterns and medium or high error levels it requires a huge amount of memory. When this happens, a lot of states are generated and never reused. On our current scheme, those states are never deallocated. In [Kur96] it is proposed that only a fixed set of nodes (the first departing from the initial state) are computed and the rest relies on dynamic programming. This may behave badly if the selection of the set of states is not adequate for the current text. We propose a more dynamic version now, which adapts to the real text. None of these ideas have been implemented yet.

When we need to create a new state and the memory limit has been reached, we select a *victim* state and, instead of allocating new space for the new state, we reuse the space of the victim state. Two issues to solve are how to select the victim and how to perform the replacement inside a graph. If we have enough memory to store the most used states, the effect of eliminating those states rarely used will be negligible.

#### 6.4.5.1 Victim Selection

This is an issue that appears on algorithms for virtual memory management, and we can borrow their policies too. In particular we believe that selecting the least recently used (LRU) state could give good results. In our case, "using" a state means being at it at some moment. This policy can be implemented by keeping an implicit doubly-linked list comprising all the states of the automaton. Each time a state is traversed, we put it at the end of the list. This involves updating 5 pointers, which unfortunately is quite expensive since it is done for each text character. An alternative is to keep, for each node, the last text position where the node was traversed (i.e. each time we traverse a node we reset the value). This works much less per character, although to select the victim we need to perform a linear pass over all the states, which is expensive. This cost could be alleviated if we eliminate a large number of rarely used nodes, so that this operation is infrequent. For instance, if we eliminate a constant fraction of the states, the extra cost of the linear search is $O(1)$ per character (amortized).

Other policies such as least frequently used (LFU) are possible. In this case we keep a counter of the number of times we traversed a state. However, an aging policy should ideally be used so that states heavily used in the past can be deallocated in the future. The process of victim selection needs also a linear search over the nodes.

#### 6.4.5.2 Victim Replacement

Besides reusing the memory space allocated for the victim, we must make sure that the graph remains consistent. We remove all the transitions leaving the victim state. This can make unreachable the target nodes of those transitions. This is not a problem since we have a separate search structure to find nodes (hash table or trie, as explained before), and we could find them later to connect them to the graph. If, on the other hand, we prefer to eliminate them, then standard garbage collection techniques can be applied to determine their reachability, namely keeping a counter of incoming transitions which if reaches zero means that the state is no longer reachable.

Notice that the cascade elimination of unreachable states may end up removing recently used states. On the other hand, LFU is more resistant to this problem, because in general a state made unreachable because of the elimination of the only state leading to it has been used less frequently than its predecessor.

We must also eliminate those transitions that lead to the victim state. In our current scheme those transitions are hard to find. A first alternative is that each state keeps a list of incoming transitions. A second one is that all states and transitions store their creation time (where "time" means the position in the text). Under normal operation, the creation time of a transition is larger than that of its target state. When a victim is replaced by a new state, however, this ceases to be true for the transitions that lead to the removed state. Hence, we do not immediately remove the incoming transitions. Rather, we defer removal to the time of normal operation: before following any transition, we check that the creation times are appropriate. Otherwise we know that the target state has been replaced and therefore remove the transition. In general, keeping the list of incoming states does not require much more space and does not degrade the normal operation of the automaton.

# Chapter 7

# Multiple Patterns

The problem of approximately searching a set of patterns (i.e. finding the occurrences of all them) has been considered only recently. A trivial solution to the multipattern search problem is to perform $r$ searches, and our aim is to search more efficiently. We present three new algorithms for on-line multipattern matching allowing errors. These are extensions of previous algorithms that search for a single pattern. The average running time achieved is in all cases linear in the text size for moderate error level, pattern length and number of patterns. They adapt (with higher costs) to the other cases. However, the algorithms differ in speed and thresholds of usefulness. We analyze theoretically when each algorithm should be used, and show experimentally their performance. The only previous solution for this problem allows only one error [MM96]. Our algorithms are the first ones to allow more than one error, and are faster than [MM96] for fewer than 50–150 patterns, depending on the parameters of the problem.

The results of this work have appeared in [BYN97b, Nav97a, BYN98e].

Many of the ideas we propose here can be used to adapt other single-pattern approximate searching algorithms to the case of multipattern matching. For instance, the idea of superimposing automata (Section 7.1) can be adapted to most bit-parallel algorithms, such as [Mye98]. Another fruitful idea is that of exact partitioning (Section 7.2), where a multipattern exact search is easily adapted to search the pieces of many patterns. There are many other filtering algorithms of the same type, e.g. [ST96].

Since the algorithms presented are extensions of on-line algorithms explained in previous chapters, the reader will frequently find backward pointers. We did our best to find a good compromise between self-containment and non-repetitiveness.

## 7.1 Superimposed Automata

In this section we describe an approach based on the bit-parallel simulation of an NFA which is able to search the pattern allowing errors. This algorithm searches for a single pattern and is developed in Chapter 5. The only connection between the search algorithm and the pattern is given by a table $b[c]$, whose $i$-th bit is zero if and only if $P_i = c$. As explained in Chapter 5, the $b[\,]$ table mechanism allows to have a set of characters at each pattern position, instead of just a single character. It suffices to set $b[c]$ to "match" at position $i$ for every $c \in P_i$. We use this property to search for multiple patterns.

Section 5.6 describes in detail how we "superimpose" the NFA's of the $r$ patterns we want to search, so we do not repeat the argument here[1]. We just recall that we build an automaton which searches a much more relaxed pattern, that is, at position $i$ it matches with the $i$-th letter of any of the patterns. Hence, it finds any of our patterns and many others which are not of interest. When the automaton finds a match, we have to verify which of the patterns have matched, if any, since the superimposed automaton does not allow us to know it.

A simple verification alternative (which we call "plain") is that once a superimposed automaton reports a match, we try the individual patterns one by one in the suspicious area. However, a smarter verification technique (which we call "hierarchical") is described in Section 4.3.2 (do not confuse with the hierarchical verification presented in Section 4.3.1). Although in Chapter 5 we used this verification to simplify the analysis, we did not implement it because in practice we superimposed few patterns and the impact would be minimal. In this chapter we have implemented it and present later experimental results.

If the number of patterns is too large, the filter will be very relaxed and will trigger too many verifications. In that case, we partition the set of patterns into groups of $r'$ patterns each, build the automaton of each group and perform $\lceil r/r' \rceil$ independent searches. The cost of this search is $O(r/r'\ n)$, where $r'$ is small enough to make the cost of verifications negligible on average. This $r'$ always exists, since for $r' = 1$ we have a single pattern per automaton and no verification is needed (indeed, the process has degenerated into sequential searching).

As in the original algorithm, we can handle classes of characters in the patterns.

### 7.1.1 Handling Longer Patterns

If the length of the patterns does not allow to put their automata in single computer words (i.e. $(m-k)(k+2) > w$), we partition the problem. We adapt the two partitioning techniques explained in Chapter 5.

**Automaton Partitioning:** if the automaton does not fit in a single word, we can partition it using a number of machine words for the simulation: once the (large) automata have been superimposed, we partition the automaton into a matrix of subautomata, each one fitting in a computer word.

Once the automaton is partitioned, we run it over the text updating its subautomata. Each step takes time proportional to the number of cells to update, i.e. $O(k(m-k)/w)$. Recall, however, that it is not necessary to update all the subautomata, since those on the right may not have any active state. We keep track of up to where need we to update the matrix of subautomata, working only on "active" cells.

The technique of grouping in case of a very relaxed filter is used here too. We use the heuristic of sorting the patterns and packing neighbors in the same group, trying to have the same first characters.

**Pattern Partitioning:** is based on the Partitioning Lemma proved in Section 4.2. We can reduce the size of the NFA's if we divide the pattern in $j$ parts, provided we search all the sub-patterns with $\lfloor k/j \rfloor$ errors. Each match of a sub-pattern must be verified to determine if it is in fact a complete match.

[1]This mechanism is used in Chapter 5 to speed up pattern partitioning, while here we use it as a truly multipattern search algorithm.

Once we partition all the patterns, we are left with $j \times r$ subpatterns to be searched with $\lfloor k/j \rfloor$ errors. We simply group them as if they were independent patterns to search with the general method. The only difference is that we have to verify the complete patterns when we find a sub-pattern.

As said in Section 5.5, we apply hierarchical verification (which is explained in Section 4.3.1) on the pattern pieces to grow up the complete pattern. This is similar to the hierarchical verification on a set of patterns we are proposing here, but it works bottom-up instead of top-down and it operates on pieces of the pattern rather than on sets of patterns. That is, instead of checking the complete pattern we check the concatenation of two pieces containing the one that matched, and if it matches then we check the concatenation of four pieces, and so on.

As we are using our hierarchical verification on the sets of pattern pieces to determine which piece matched given that a superimposition of them matched, we are coupling two different hierarchical verification techniques in this case: we first use our new mechanism to determine which piece matched from the superimposed group and then use the technique of Section 4.3.1 to determine the occurrence or not of the complete pattern the piece belongs to.

## 7.2 Partitioning into Exact Searching

This technique (called "exact partitioning" for short) is based on a single-pattern filter which reduces the problem of approximate searching to a problem of multipattern exact searching. The single-pattern algorithm is explained in Section 6.1.

The idea is based on a particular case of the Partitioning Lemma which states that if we partition the pattern in $k+1$ pieces, then at least one piece must appear unchanged in any occurrence with $k$ errors or less. Since there are efficient algorithms to search for a set of patterns exactly, we partition the pattern in $k+1$ pieces (of similar length), and apply a multipattern exact search for the pieces. Each occurrence of a piece is verified to check if it involves a complete match. If there are not too many verifications, this algorithm is extremely fast. We use the same technique for hierarchical verification of a single pattern presented in Section 4.3.1.

We can easily add more patterns to this scheme. Suppose we have to search for $r$ patterns $P^1, ..., P^r$. We cut each one into $k+1$ pieces and search in parallel for *all* the $r \times (k+1)$ pieces. When a piece is found in the text, we use a classical algorithm to verify its pattern in the candidate area (this time we normally know which pattern to verify, since we know which piece matched). As for superimposed automata, this constitutes a good filter if the number of patterns and errors is not too high. Unlike superimposed automata, grouping and hierarchical verification are of no use here, since there are no more matches in the union of patterns than the sum of the individual matches. The only reason to superimpose fewer patterns is that the shifts of a multipattern Boyer-Moore-like algorithm are reduced as the number of patterns grow, but as we show later this is not important in practice (recall that we use an extension of the Sunday [Sun90] algorithm).

## 7.3 A Counting Filter

We present now a filter based on counting letters in common between the pattern and a text window. The single-pattern version of this filter is presented in Section 6.2. The idea is to slide a window of length $m$ over the text and keep track of how many letters of the window are present in the pattern, triggering a verification when there are $m-k$ matches or more. The code is very simple and is shown in Figure 6.8. It is based on keeping a counter *count* of how many characters in the current text window match the pattern, and a table $A[c]$ which for each character tells how many characters of the text window can currently be taken as part of the pattern. Actually, we keep $counter - (m - k)$, so when it reaches zero we trigger a verification.

To search $r$ patterns in the same text, we use bit-parallelism to keep all the counters in a single machine word. We must do that for the $A[\,]$ table and for *count*.

The values of the entries of $A[\,]$ lie in the range $[-m..m]$, so we need exactly $1 + \lceil \log_2(m+1) \rceil$ bits to store them. This is also enough for *count*, since it is in the range $[-(m-k)..k]$. Hence, we can pack

$$\left\lfloor \frac{w}{1 + \lceil \log_2(m+1) \rceil} \right\rfloor$$

patterns of length $m$ in a single search (recall that $w$ is the number of bits in the computer word). If the patterns have different lengths, this limit holds for the longest one. If we have more patterns, we must divide the set in subsets of at most this size and search each subset separately. We focus our attention on a single subset now.

The algorithm simulates the simple one as follows. We have a table $MA[\,]$ that packs all the $A[\,]$ tables. Each entry of $MA[\,]$ is divided in bit areas of the appropriate length. In the area of the machine word corresponding to each pattern, we store its normal $A[\,]$ value, set to 1 the most significant bit of the area, and subtract 1. When, in the algorithm, we have to add or subtract 1, we can easily do it in parallel without causing overflow from an area to the next. Moreover, the corresponding $A[\,]$ value is not positive if and only if the most significant bit of the area is zero. Figure 7.1 illustrates.

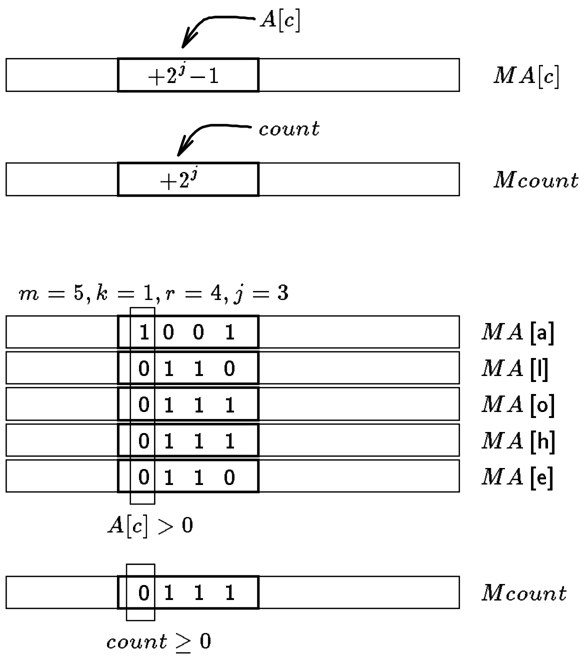


Figure 7.1: Scheme and an example of the bit-parallel counters. The example follows that of Figure 6.7.

We have a parallel counter *Mcount*, where the areas are aligned with $MA[\,]$. It is initialized by setting to 1 the most significant bit of each area and then subtracting $m-k$ at each one. Later, we can add or subtract 1 in parallel without causing overflow. Moreover, the window must be verified for a pattern whenever the most significant bit of its area reaches 1. The condition can be checked in parallel, although if some counter reaches zero we sequentially verify which one did it. Note that this allows to have different $k$ values for each pattern. It is also possible to have different $m$ values, but the performance of the algorithm may be degraded if they are very different, because we have to use the longest text window for all the patterns, and the width of the bit areas corresponds to the longest pattern.

Finally, observe that the counters that we want to selectively increment or decrement correspond exactly to the $MA[\,]$ areas that have a 1 in their most significant bit (i.e. those whose $A[\,]$ value is positive). This allows an obvious bit mask-shift-add mechanism to perform this operation in parallel on all the counters.

Figure 7.2 shows the pseudocode of the parallel algorithm. As it can be seen, the algorithm is more complex than the simple version but the number of operations per character is still very low.

```
CountFilter ($T$,$n$,$P^{1..r}$,$m$,$k$)

{        /* preprocessing */
   $\ell = \lceil \log_2 m \rceil$;
   for ($c \in \Sigma$) $MA[c] = (01^\ell)^r$;
   for ($s = 0; s < r; s{+}{+}$)
      for ($i = 1; i \leq m; i{+}{+}$) $MA[P_i^{s+1}]$ $+=$ $10^{s(\ell+1)}$;
   $high = (10^\ell)^r$;
   $ones = (0^\ell 1)^r$;
   $Mcount = (10^\ell - (m-k)) \times ones$;

         /* searching */
   for ($j = 1; j \leq m; j{+}{+}$)       /* fill the initial window */
    { $Mcount$ $+=$ $(MA[T_j] >> \ell)$ & $ones$;
      $MA[T_j]$ $-=$ $ones$;
    }
   for ($; j \leq n; j{+}{+}$)       /* move the window */
    { if ($Mcount$ & $high \neq 0$) then      /* verify the area */
          verify $T_{j-m..j-1}$ with dynamic programming
            (for each pattern whose high $Mcount$ bit is 1)
      $MA[T_{j-m}]$ $+=$ $ones$;
      $Mcount$ $-=$ $(MA[T_{j-m}] >> \ell)$ & $ones$;
      $Mcount$ $+=$ $(MA[T_j] >> \ell)$ & $ones$;
      $MA[T_j]$ $-=$ $ones$;
    }
}
```

Figure 7.2: The code of our multiple-pattern algorithm, in C-like notation.

As in the original algorithm, we can handle classes of characters in this scheme.

## 7.4 Analysis

We are interested in the complexity of the presented algorithms, as well as in the restrictions that $\alpha$ and $r$ must satisfy for each mechanism to be efficient in filtering most of the unrelevant part of the text.

To this effect, we define two concepts. First, we say that a multipattern search algorithm is *optimal* if it searches $r$ patterns in the same time it takes to search one pattern. If we call $C_{n,r}$ the cost to search $r$ patterns in a text of size $n$, then an algorithm is optimal if $C_{n,r} = C_{n,1}$. Second, we say that a multipattern search algorithm is *useful* if it searches $r$ patterns in less than the time it takes to search them one by one with the corresponding sequential algorithm, i.e. $C_{n,r} < r\,C_{n,1}$. As we work with filters, we are interested in the average case analysis, since in the worst case none is useful.

We compare in Table 7.1 the complexities and limits of applicability of all the algorithms. The analysis leading to these results is presented later in this section.

| Algorithm | Complexity | Optimality | Usefulness |
|---|---|---|---|
| Simple Superimp. | $\frac{r}{\sigma(1-\alpha)^2}\ n$ | $\alpha < 1 - e\sqrt{\frac{r}{\sigma}}$ | $\alpha < 1 - e/\sqrt{\sigma}$ |
| Automaton Part. | $\frac{\alpha m^2 r}{\sigma w(1-\alpha)}\ n$ | $\alpha < 1 - e\sqrt{\frac{r}{\sigma}}$ | $\alpha < 1 - e/\sqrt{\sigma}$ |
| Pattern Part. | $\frac{mr}{\sigma\sqrt{w}(1-\alpha)}\ n$ | $\alpha < 1 - e\sqrt{\frac{r}{\sigma}}$ | $\alpha < 1 - e/\sqrt{\sigma}$ |
| Part. Exact Search | $\left(1 + \frac{rm}{\alpha\sigma^{1/\alpha}}\right)\ n$ | $\alpha < \frac{1}{\log_\sigma(rm)+\Theta(\log_\sigma \log_\sigma(rm))}$ | $\alpha < \frac{1}{\log_\sigma m+\Theta(\log_\sigma \log_\sigma m)}$ |
| Counting | $\frac{r \log m}{w}\ n$ | $\alpha < e^{-m/\sigma}$ | $\alpha < e^{-m/\sigma}$ |
| Muth & Manber | $mn$ | $k = 1$ | $k = 1$ |

Table 7.1: Complexity, optimality and limit of applicability for the different algorithms.

We present in Figure 7.3 a schematical representation of the areas where each algorithm is the best in terms of complexity. We show later that the experiments confirm these figures. Notice that our limits come in fact from big-$O$ expressions, so they only give an idea of the real scenario.

- Exact partitioning is the fastest choice in most reasonable scenarios, for the error levels where it can be applied. First, it is faster than counting for $m/\log m < \alpha\sigma^{1/\alpha}/w$, which does not hold asymptotically but holds in practice for reasonable values of $m$. Second, it is faster than superimposing automata for $\min(\sqrt{w}, w/m) < \sigma^{1/\alpha-1}/(1/\alpha - 1)$, which is true in most practical cases.

- The only algorithm which can be faster than exact partitioning is that of Muth & Manber [MM96], namely for $r > \alpha\sigma^{1/\alpha}$. However, it is limited to $k = 1$.

- For increasing $m$, counting is asymptotically the fastest algorithm since its cost grows as $O(\log m)$ instead of $O(m)$ thanks to its optimal use of the bits of the computer word. However, its applicability is reduced as $m$ grows, being in practice useless at the point where it wins over exact partitioning.

- When the error level is too high for exact partitioning, superimposing automata is the only remaining alternative. Automaton partitioning is better for $m \leq \sqrt{w}$, while pattern partitioning is asymptotically better. Both algorithms have the same limit of usefulness, and for higher error levels no filter can improve over a sequential search.

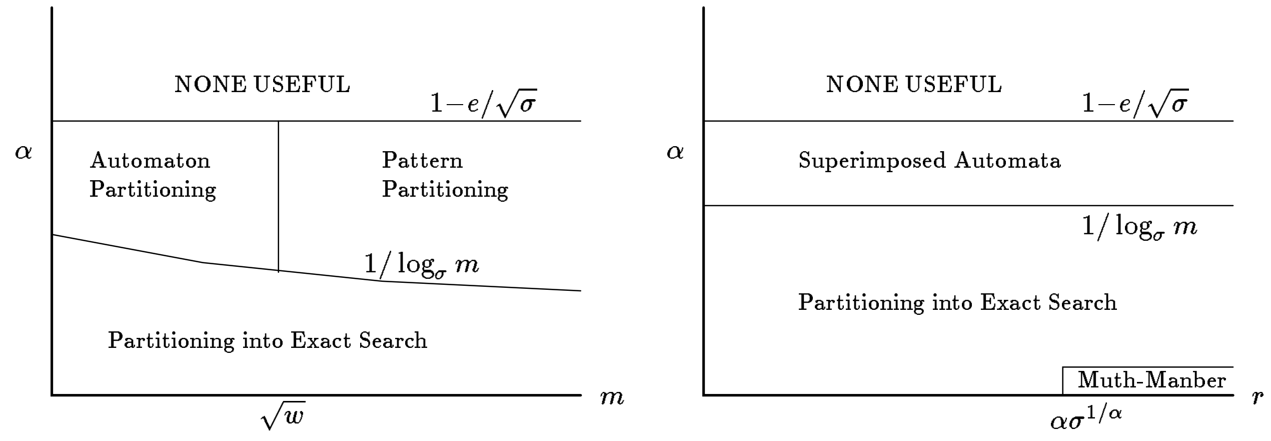


Figure 7.3: The areas where each algorithm is better, in terms of $\alpha$, $m$ and $r$. In the left plot (varying $m$), we have assumed a moderate $r$ (say, less than 50).

We detail now the analysis of each algorithm.

### 7.4.1 Superimposed Automata

Suppose that we search $r$ patterns. As explained before, we can partition the set in groups of $r'$ patterns each, and search each group separately (with its $r'$ automata superimposed). The size of the groups should be as large as possible, but small enough for the verifications to be not significant. As shown in Section 5.7.4, the matching probability is the same as for a single pattern, provided we replace $\sigma$ by $\sigma/r$.

As the single-pattern algorithm is $O(n)$ time, the multipattern algorithm is optimal on average whenever the total cost of verifications is $O(1)$ per character. Since each verification costs $O(m)$ (because we use a linear-time algorithm on an area of length $m + k = O(m)$), we need that the total number of verifications performed is $O(1/m)$ per character, on average. If we used the plain verification scheme, this would mean that the probability that a superimposed automaton matches a text position should be $O(1/(mr))$, as we have to perform $r$ verifications.

In a previous version of this work [BYN97b], where hierarchical verification was not used, we analyzed the cost of verifications. In that case, we had that as $r$ was increased, matching was more probable (because it was easier to cross a horizontal edge of the automaton) *and* it costed more (because we had to check the $r$ patterns one by one). The results was that there were two different limits on the maximum allowable $r$, one for each of the two facts just stated. The limit due to the increased cost of each verification was more stringent than that of increased matching probability. Moreover, the resulting analysis was very complex.

This improves considerably with hierarchical verification. As we show in Section 4.3.2, the average cost to verify a match of the superimposed automaton is $O(m)$ when hierarchical verification is used, instead of the $O(rm)$ cost of plain verification. That is, the cost does *not* grow as the number of patterns increases.

Hence, the only limit that prevents us from superimposing all the $r$ patterns is that the matching probability becomes higher. That is, if $\alpha > 1 - e\sqrt{r/\sigma}$, then the matching probability is too high and we will spend too much time verifying almost all text positions. On the other hand, we can superimpose as much as we like before that limit is reached. This tells that the best $r$ (which we call $r^*$) is the maximum one not reaching the limit, i.e.

$$r^* \;=\; \frac{\sigma(1-\alpha)^2}{e^2} \tag{7.1}$$

which matches the result obtained in Section 5.7.4.

Since we partition in sets small enough to make the verifications not significant, the cost is simply $O(r/r^*\,n) = O(rn/(\sigma(1-\alpha)^2))$.

This means that the algorithm is optimal for $r = O(\sigma)$ (taking the error level as a constant), or alternatively $\alpha \leq 1 - e\sqrt{r/\sigma}$. On the other hand, for $\alpha > 1 - e/\sqrt{\sigma}$, the cost is $O(rn)$, not better than the trivial solution (i.e. $r^* = 1$ and hence no superimposition occurs and the algorithm is not useful). Figure 7.4 illustrates.

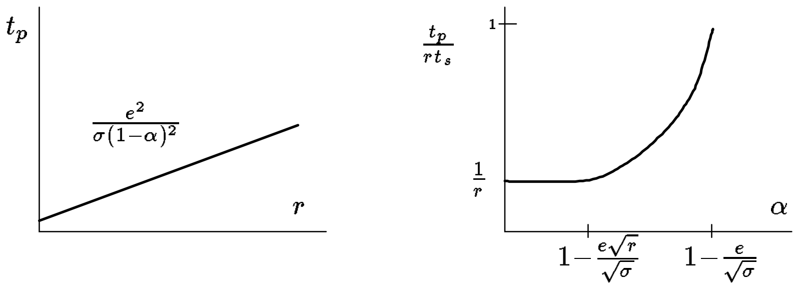


Figure 7.4: Behavior of superimposed automata. On the left, the cost increases linearly with $r$, with slope depending on $\alpha$. On the right, the cost of a parallel search ($t_p$) approaches $r$ single searches ($rt_s$) when $\alpha$ grows.

**Automaton Partitioning:** the analysis for this case is similar to the simple one, except because each step of the large automaton takes time proportional to the total number of subautomata, i.e. $O(k(m-k)/w)$. In fact, this is a worst case since on average not all cells are active, but we use the worst case because we superimpose all the patterns we can until the worst case of the search is almost reached. Therefore, the cost formula is

$$\frac{e^2}{(1-\alpha)^2\sigma}\,\frac{k(m-k)}{w}\,rn \;=\; O\left(\frac{\alpha m^2}{\sigma w(1-\alpha)}\,rn\right)$$

This is optimal for $r = O(\sigma w)$ (for constant $\alpha$), or alternatively for $\alpha \leq 1 - e\sqrt{r/\sigma}$. It is useful for $\alpha \leq 1 - e/\sqrt{\sigma}$.

**Pattern Partitioning:** we have now $jr$ patterns to search with $\lfloor k/j \rfloor$ errors. The error level is the same for subproblems (recall that the subpatterns are of length $m/j$).

To determine which piece matched from the superimposed group, we pay $O(m)$ independently of the number of pieces superimposed (thanks to the hierarchical verification). Hence the limit for our grouping is given by Eq. (7.1). In both the superimposed and in the single-pattern algorithm, we also pay to verify if the match of the piece is part of a complete match. As we show in Section 4.3.1, this cost is negligible for $\alpha < 1 - e/\sqrt{\sigma}$, which is less strict than the limit given by Eq. (7.1).

As we have $jr$ pieces to search, the complexity is

$$\frac{jre^2}{\sigma(1-\alpha)^2}\, n \;=\; O\left(\frac{m}{\sigma\sqrt{w}(1-\alpha)}\, rn\right)$$

(recall that $j = O((m-k)/\sqrt{w})$, from Eq. (5.7) and the development that follows).

On the other hand, the search cost of the single-pattern algorithm is $O(jrn)$. With respect to the simple algorithm for short patterns, both costs have been multiplied by $j$, and therefore the limits for optimality and usefulness are the same.

If we compare the complexities of pattern versus automaton partitioning, we have that pattern partitioning is better for $k > \sqrt{w}$. This means that for constant $\alpha$ and increasing $m$, pattern partitioning is asymptotically better.

### 7.4.2 Partitioning into Exact Searching

In Section 6.1 we analyze this algorithm as follows. Except for verifications, the search time can be made $O(n)$ in the worst case by using an Aho-Corasick machine [AC75], and $O(\alpha n)$ in the best case if we use a multipattern Boyer-Moore algorithm. This is because we search pieces of length $m/(k+1) \approx 1/\alpha$.

We are interested in analyzing the cost of verifications. Since we cut the pattern in $k+1$ pieces, they are of length $\lfloor m/(k+1)\rfloor$ or $\lceil m/(k+1)\rceil$. The probability of each piece matching is at most $1/\sigma^{\lfloor m/(k+1)\rfloor}$. Hence, the probability of any piece matching is at most $(k+1)/\sigma^{\lfloor m/(k+1)\rfloor}$.

We can easily extend that analysis to the case of multiple search, since we have now $r(k+1)$ pieces of the same length. Hence, the probability of verifying is $r(k+1)/\sigma^{\lfloor m/(k+1)\rfloor}$. We check the matches using a classical algorithm such as dynamic programming. Note that in this case we know which pattern to verify. As we show in Section 4.3.1, the total verification cost if the pieces are of length $\ell$ is $O(\ell^2)$ (in our case, $\ell = m/(k+1)$). Hence, the search cost is

$$O\left(1 + \frac{rm}{\alpha\sigma^{1/\alpha}}\right)\, n$$

where the "1" must be changed to "$\alpha$" if we consider the best case of the search.

We consider optimality and usefulness now. An optimal algorithm should pay $O(n)$ total search time, which holds for

$$\alpha \;<\; \frac{1}{\log_\sigma(rm) + \log_\sigma(1/\alpha)} \;=\; \frac{1}{\log_\sigma(rm) + \Theta(\log_\sigma \log_\sigma(rm))}$$

where the last equality is obtained by noticing that $\alpha < 1/(3\log_\sigma m + \log_\sigma r)$ is a pessimistic bound valid when plain verification is used. This last result applies also if we consider the best case of the search.

The algorithm is always useful, since it searches at the same cost independently on the number of patterns, and the number of verifications triggered is exactly the same as if we searched each pattern separately. However, if $\alpha > 1/(\log_\sigma m + \Theta(\log_\sigma \log_\sigma m))$, then both algorithms (single and multipattern) work as much as dynamic programming and hence the multipattern search is not useful (see Section 6.1). The other case when the algorithm could not be useful is when the shifts of a Boyer-Moore search are shortened by having many patterns up to the point where it is better to perform separate searches. This never happens in practice.

### 7.4.3 Counting

If the number of verifications is negligible, each pass of the algorithms is $O(n)$. In the case of multiple patterns, only $O(w/\log m)$ patterns can be packed in a single search, so the cost to search $r$ patterns is $O(rn\log(m)/w)$.

In Section 6.2.2.2 we obtain a pessimistic bound on the maximum error level $\alpha$ that the filtration scheme can tolerate while keeping the number of verifications low, namely $\alpha < e^{-m/\sigma}$ (Eq. (6.3)). This is in fact the limit value for $\alpha$ so that the matching probability is $O(1/m^c)$ for any constant $c > 0$.

We assume that we use dynamic programming to verify potential matches. If the probability of verifying is at most $\log(m)/(wm^2)$ then the algorithm keeps linear (i.e. optimal) on average. The algorithm is always useful since the number of verifications triggered with the multipattern search is the same as for the single-pattern version. However, if the matching probability exceeds $1/m$ then both algorithms work $O(rmn)$ as for dynamic programming and hence the filter is not useful. Therefore, $\alpha < e^{-m/\sigma}$ is both the limit of optimality and usefulness of our algorithm.

## 7.5 Experimental Results

We experimentally study our algorithms and compare them against previous work. We tested with 10 Mb of random text ($\sigma = 32$) and lower-case English text. Each data point was obtained by averaging the Unix's user time over 10 trials. We present all the times in tenths of seconds per Mb.

Figure 7.5 compares the plain and hierarchical verification methods against a sequential application of the $r$ searches, for the case of superimposed automata when the automaton fits in a computer word. Hierarchical verification clearly outperforms plain verification in all cases. Moreover, the analysis for hierarchical verification is confirmed since the maximum $r$ up to where the cost of the parallel algorithm does not grow linearly is very close to $r^* = (1-\alpha)^2\sigma/1.09^2$. On the other hand, the algorithm with simple verification degrades sooner, since the verification cost grows with $r$.

The mentioned maximum $r^*$ value is the point where the parallelism ratio is maximized. That is, if we have to search for more than $r^*$ patterns, it is better to split them in groups of size $r^*$ and search each group sequentially. To stress this point, Figure 7.6 shows the quotient between the parallel and the sequential algorithms, where the optimum is clear for superimposed automata. On the other hand, the exact partitioning algorithm does not degrade its parallelism ratio, as predicted by the analysis. When we compare our algorithms against the others, we use this $r^*$ value to obtain the optimal grouping for the superimposed automata algorithms. The exact partitioning, on the other hand, performs all the searches in a single pass. We have not included counting in this test because its degree of parallelism is fixed and cannot be controlled.

Notice that the plots which depend on $r$ show the point where $r^*$ should be selected. Those which depend in $k$ (for fixed $r$), on the other hand, just show how the parallelization works as the error level increases, which cannot be controlled by the algorithm.

Now that we have established that hierarchical verification is superior we do not consider plain verification anymore. We turn now to the problem of selecting among pattern partitioning or automaton partitioning. Figure 7.7 shows the case of patterns of length 30. As it can be seen there is no clear winner. It depends on the error level and the number of patterns.

Figure 7.8 shows the speedups, including that of exact partitioning. The picture is much more complex now. The speedup of pattern partitioning is easily predicted by noticing that it just splits

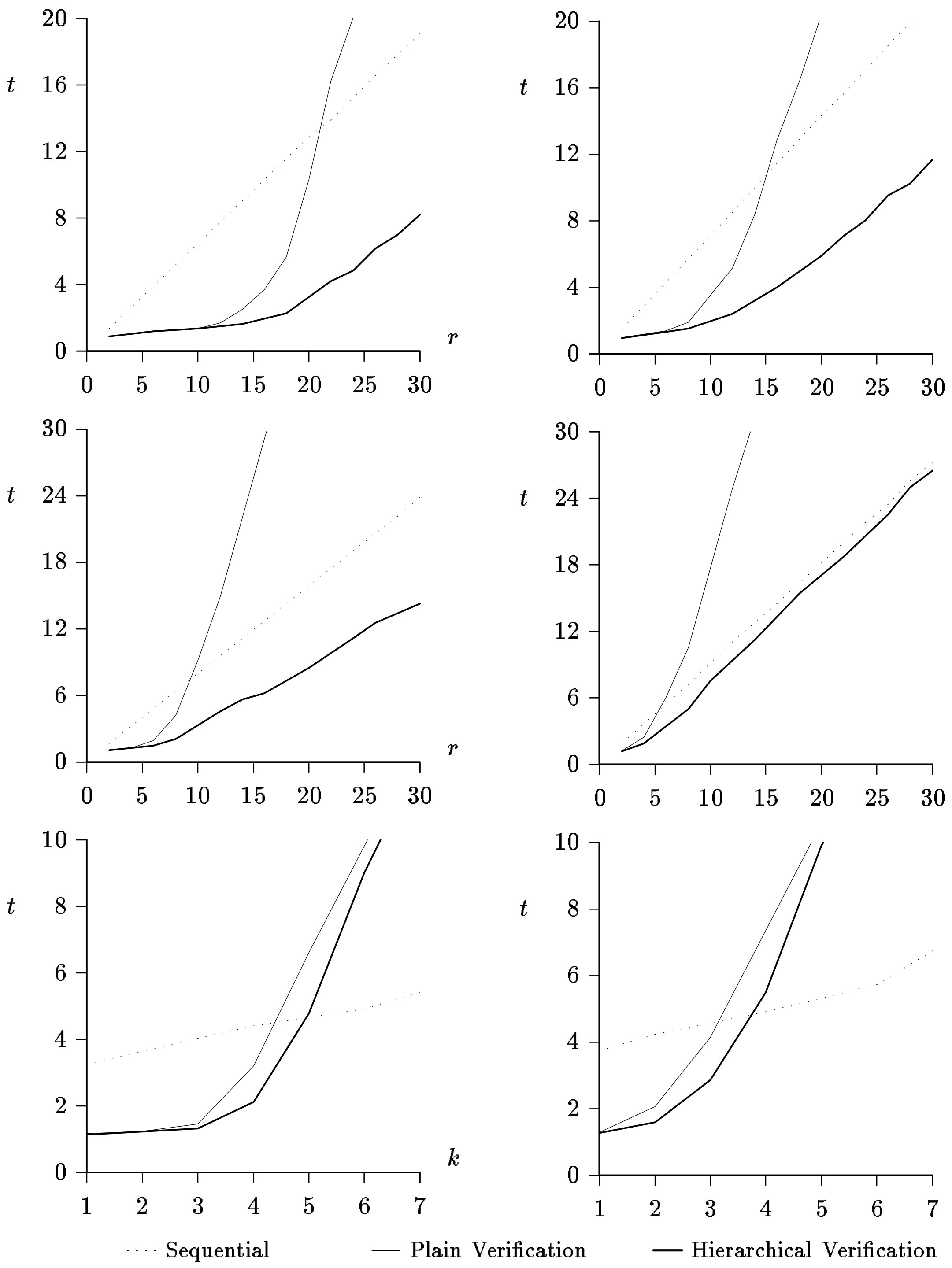


Figure 7.5: Comparison of sequential and superimposed automata for $m = 9$. The left plots are on random text and the right plots on English text. The rows correspond to $k = 1$, $k = 3$ and $r = 5$, respectively.

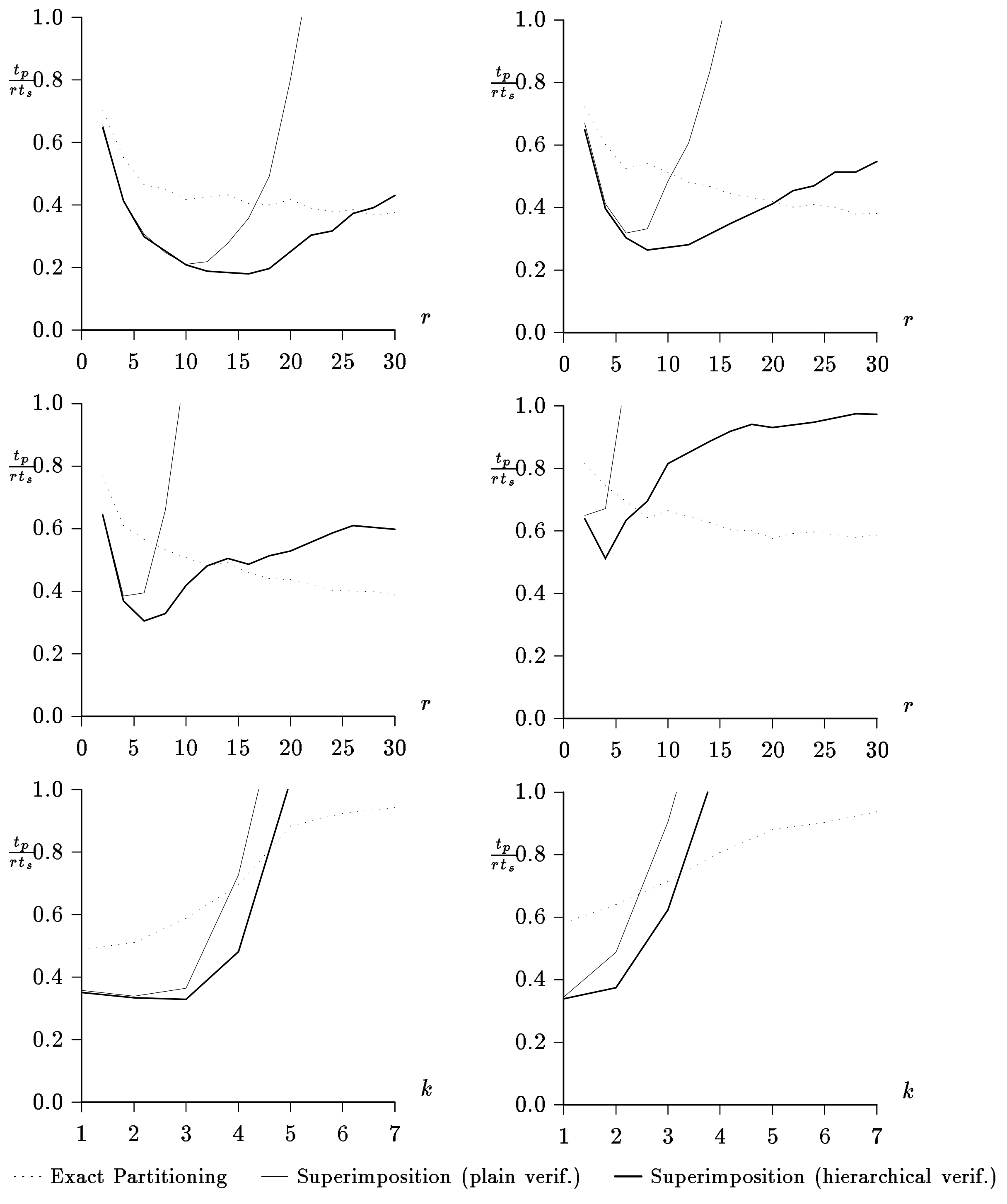


Figure 7.6: Ratio between parallel and sequential automata algorithms for $m = 9$. The left plots are on random text and the right plots on English text. The rows correspond to $k = 1$, $k = 3$ and $r = 5$, respectively.

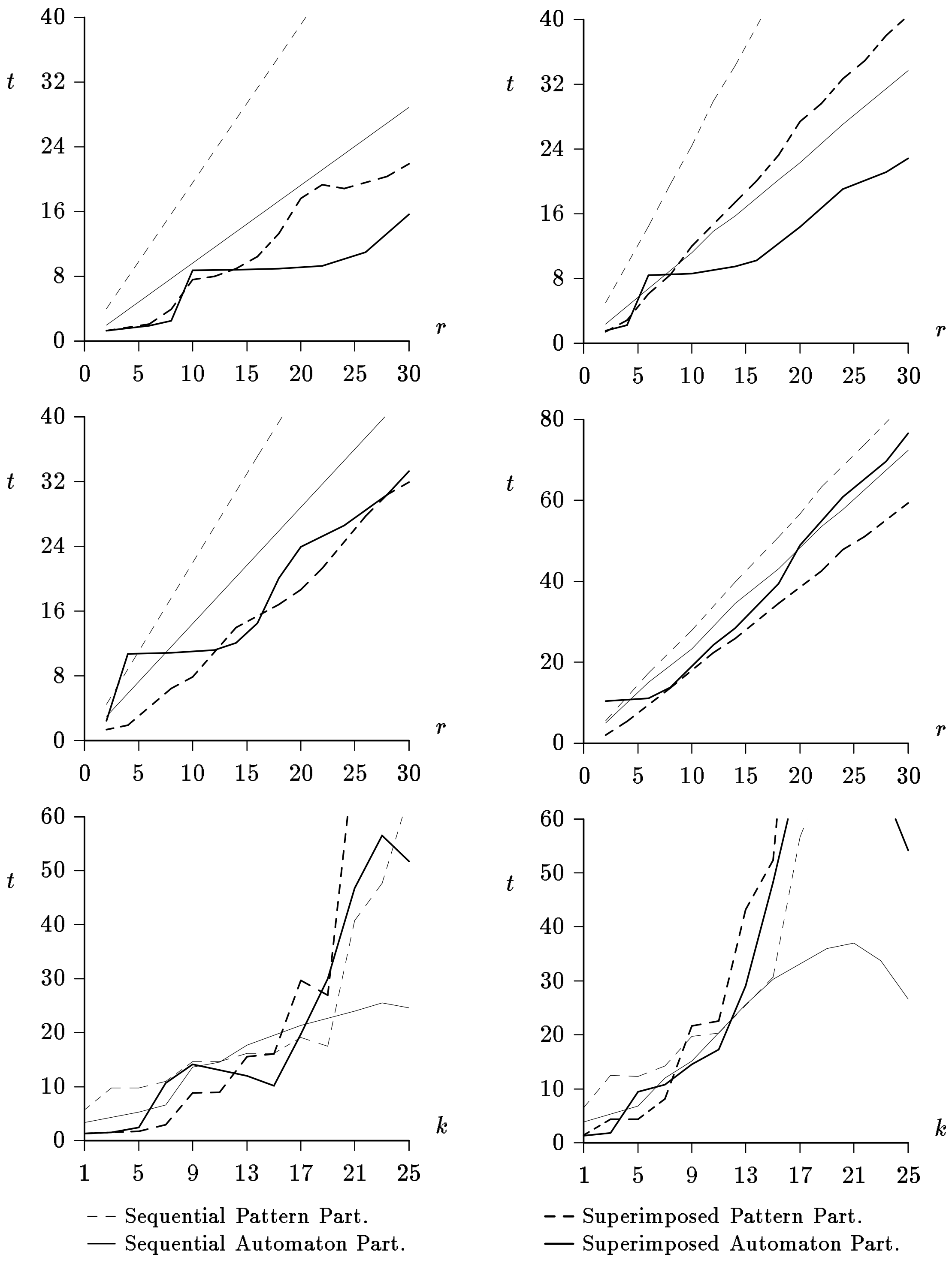


Figure 7.7: Comparison of sequential and superimposed automata algorithms for $m = 30$. The left plots are on random text and the right plots on English text. The rows correspond to $k = 4$, $k = 8$ and $r = 5$, respectively.

the pattern in small ones and resorts to the simple algorithm. That is, for $m = 30$ and $k = 4$ it splits the pattern in 4 pieces for which $m = 7$ or 8 and $k = 1$. Since the optimum $r^*$ for those single pieces is close to 16, the method gets it optimum speedup near $r^* = 4$. The case of automaton partitioning is more complex, since now the search cost is not uniform but it depends on the number of active cells of the automaton. There are local optima separated by bad combinations. Finally, there is an optimum for exact partitioning (given by the Boyer-Moore shifts). However, we checked that the difference is not so important to justify splitting a single search in two.

We compare now our algorithms among them and against others. We begin with short patterns whose NFA fit in a computer word. Figure 7.9 shows the results for increasing $r$ and Figure 7.10 for increasing $\alpha$. For low and moderate error levels, exact partitioning is the fastest algorithm. In particular, it is faster than previous work [MM96] when the number of patterns is below 50 (for English text) or 150 (for $\sigma = 32$). When the error level increases, superimposed automata is the best choice. This agrees with the analysis.

We consider longer patterns now ($m = 30$). Figure 7.11 shows the results for increasing $r$ and Figure 7.12 for increasing $\alpha$. We have a similar scenario: exact partitioning is the best where it can be applied, and improves over previous work [MM96] for $r$ up to 90–100. For these longer patterns the superimposed automata technique also degrades, and only rarely is it able to improve over exact partitioning. In most cases it only begins to be the best when it (and all the others) are no longer useful.

As it can be seen, counting is competitive but it is never the fastest algorithm. It is however simple and elegant, and its theoretical complexity is good as $m$ grows. When it was first published [Nav97a] (before hierarchical verification improved the other two algorithms) it was the fastest for intermediate error levels.

There is number of heuristic optimizations which can be done on our algorithms and which we have not pursued yet, for instance

- If the patterns have different lengths, we truncate them to the shortest one when superimposing automata. We can select cleverly the substrings to use, since having the same character at the same position in two patterns improves the filtering mechanism.
- We used simple heuristics to group subpatterns in superimposed automata. These can be improved to maximize common letters too. A more general technique could group patterns which are at small edit distance (i.e. a clustering technique).
- We are free to partition each pattern in $k + 1$ pieces as we like in exact partitioning. This is used in Section 6.1 to minimize the expected number of verifications when the letters of the alphabet do not have the same probability of occurrence (e.g. in English text). We presented an $O(m^3)$ dynamic programming algorithm to select the best partition, and this could be applied to multipattern search.

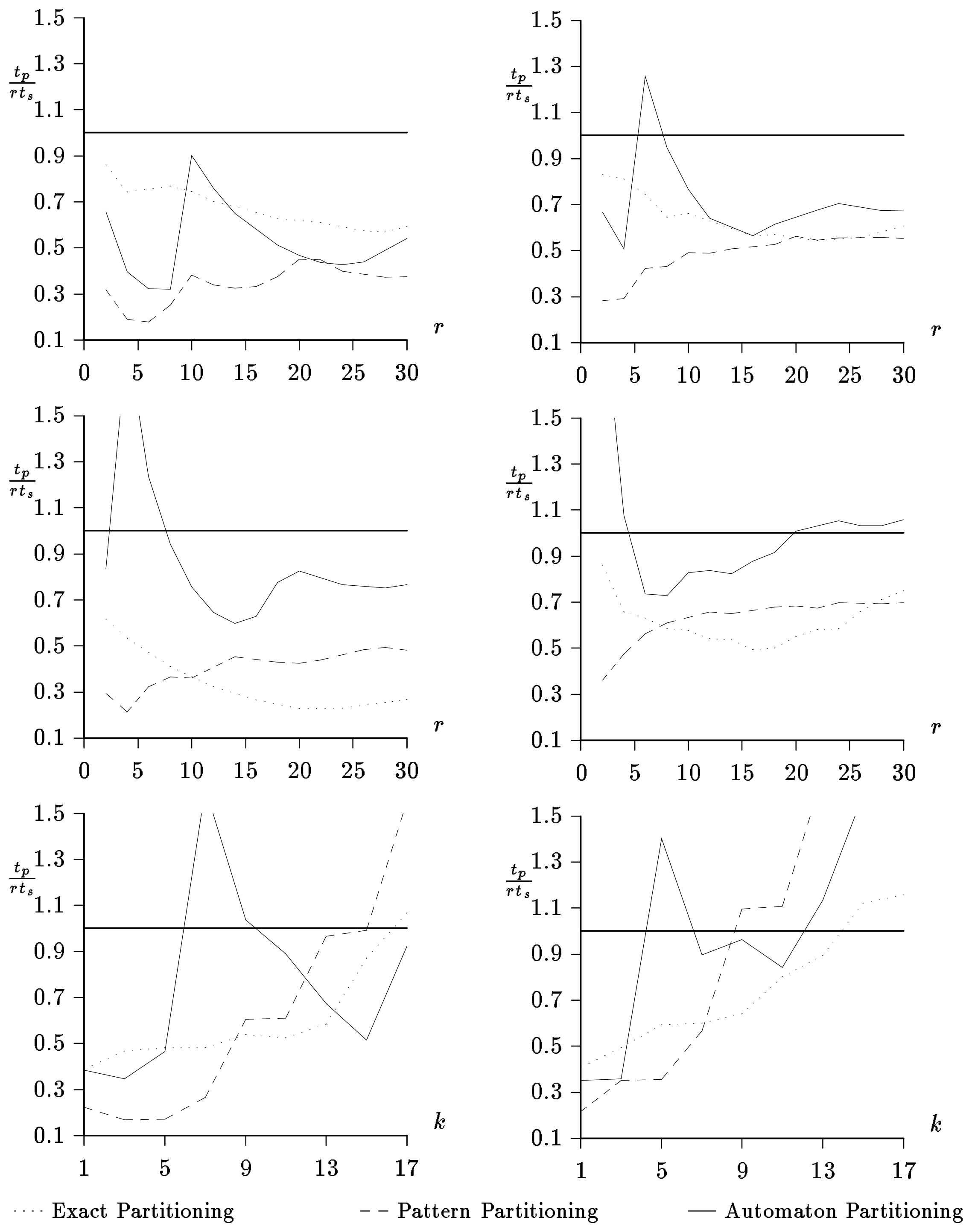


Figure 7.8: Ratio between parallel and sequential algorithms for $m = 30$. The left plots are on random text and the right plots on English text. The rows correspond to $k = 4$, $k = 8$ and $r = 5$, respectively.

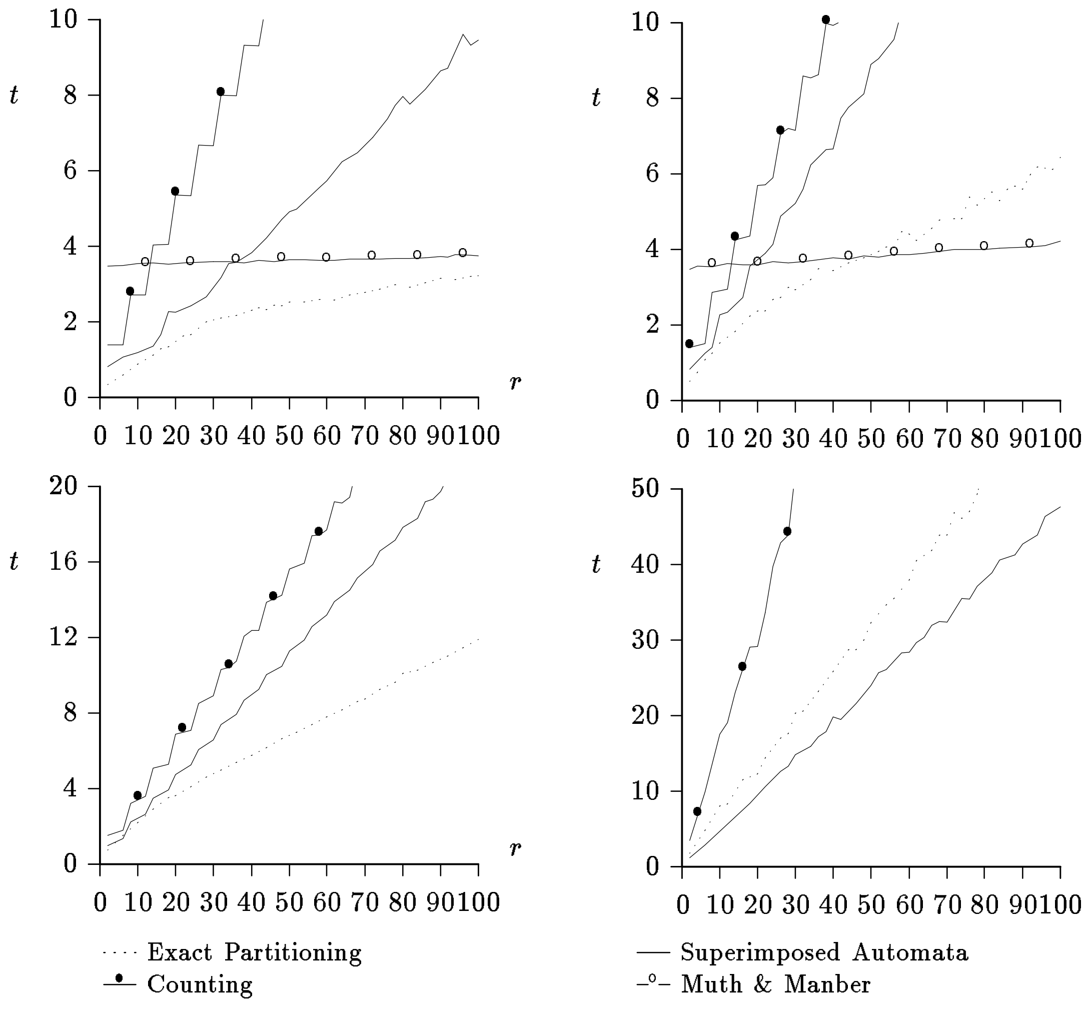


Figure 7.9: Comparison among algorithms for $m = 9$ and increasing $r$. The rows show $k = 1$ and $k = 3$, respectively. The left plot shows random text, the right one shows English text.

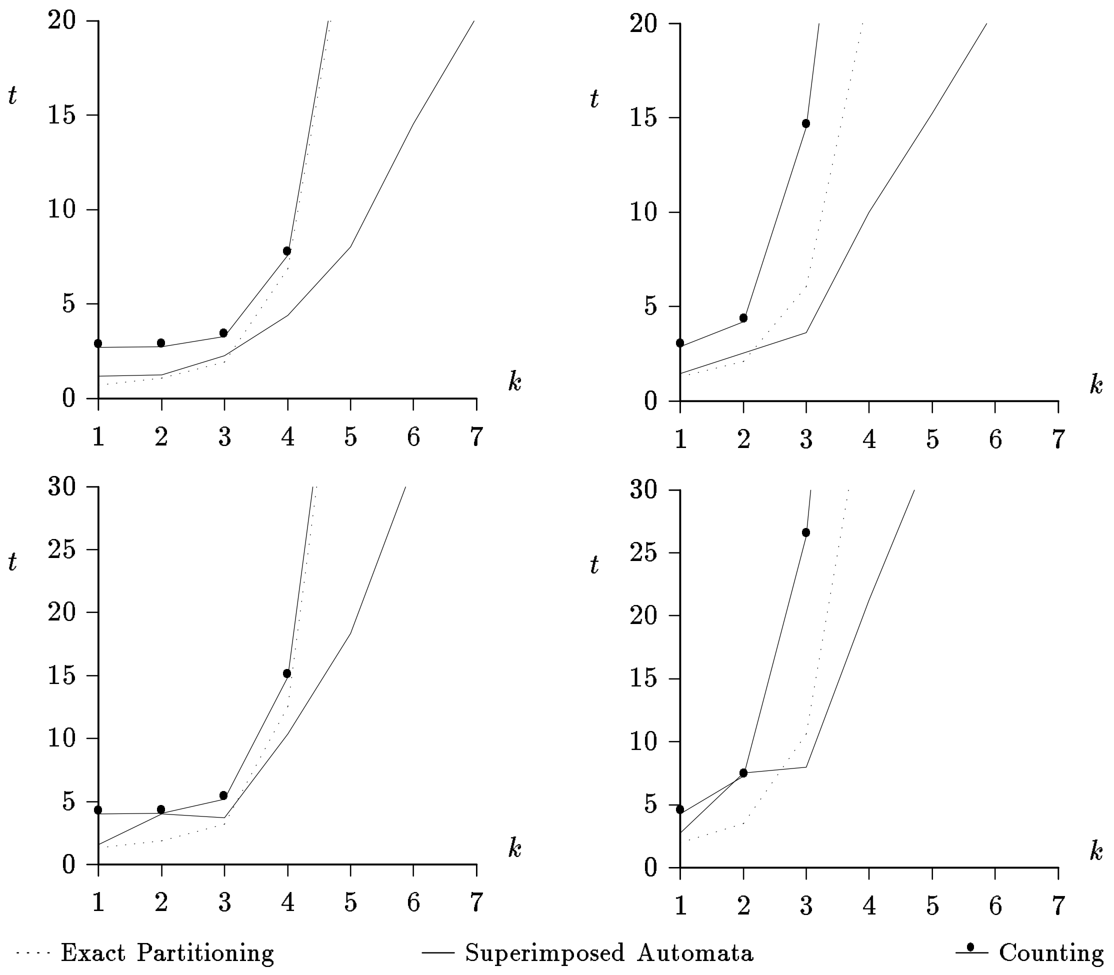


Figure 7.10: Comparison among algorithms for $m = 9$ and increasing $k$. The rows show $r = 8$ and $r = 16$, respectively. The left plot shows random text, the right one shows English text.

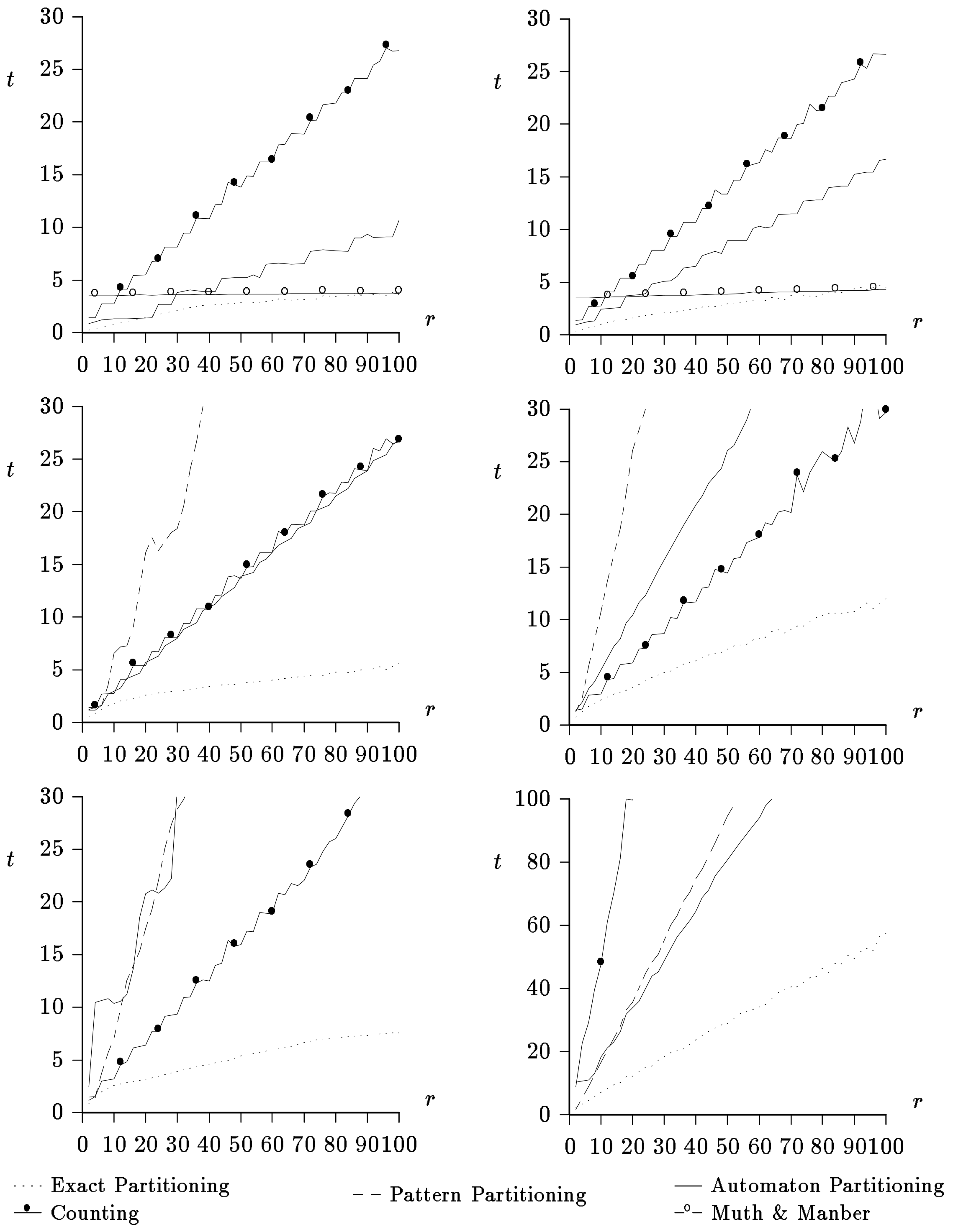


Figure 7.11: Comparison among algorithms for $m = 30$ and increasing $r$. The rows show $k = 1$, $k = 4$ and $k = 8$, respectively. The left plot shows random text, the right one shows English text. Pattern partitioning is not run for $k = 1$ because it should resort to exact partitioning.

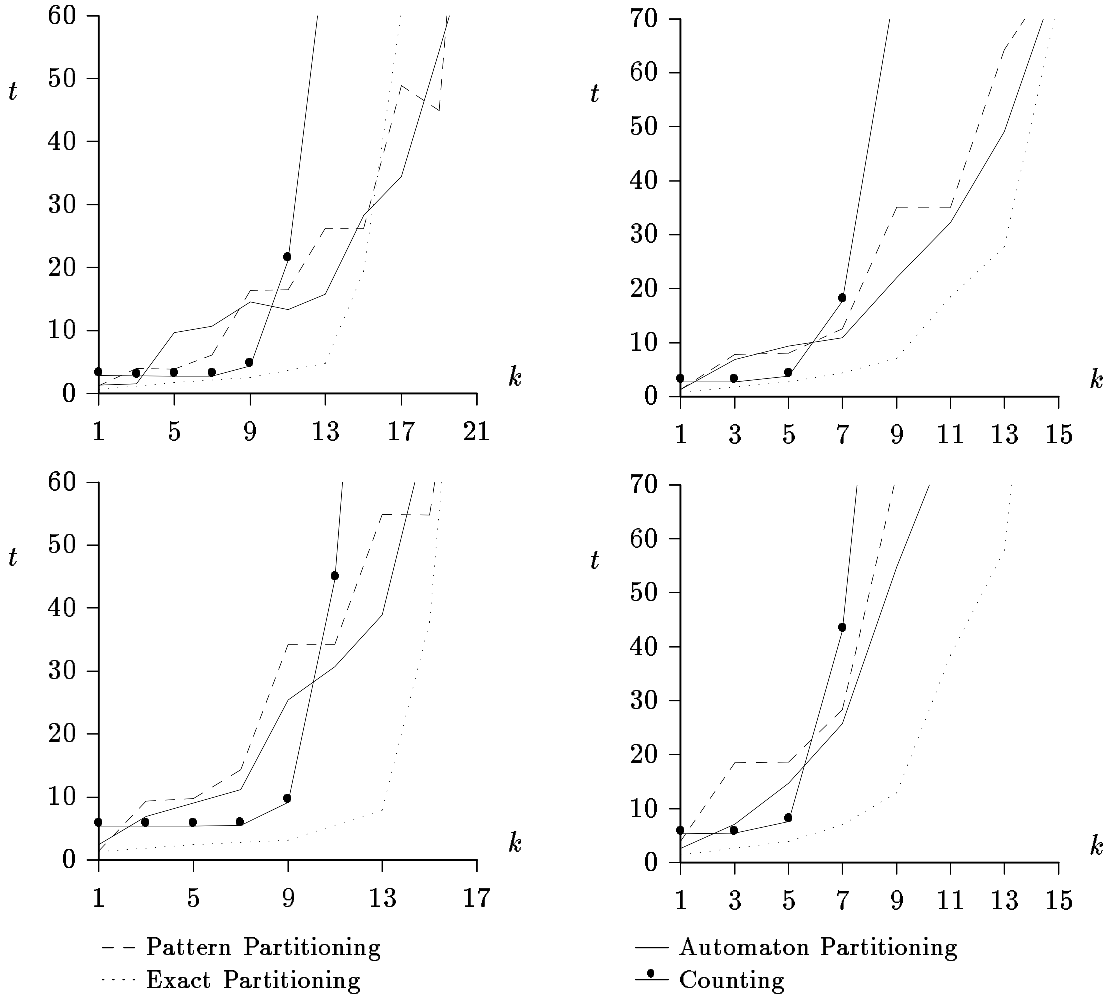


Figure 7.12: Comparison among algorithms for $m = 30$ and increasing $k$. The rows show $r = 8$ and $r = 16$, respectively. The left plot shows random text, the right one shows English text.

# Part II

# Indexed Searching

This Part of this thesis deals with indexing techniques for approximate string matching. The problem of indexing a text to answer approximate queries is quite difficult and was considered until 1992 the big open problem in this area. Since 1992, two kinds of indices have appeared. A first one ("word-retrieving") aims to index natural language text and to retrieve whole words which match the query with up to $k$ errors. The second one ("sequence-retrieving") keeps the original problem and does not assume special text characteristics.

Word-retrieving indices have been very successful in practice, with a performance approaching that of exact searching for reasonable queries. Their restriction of matching words against words is reasonable in many information retrieval scenarios, although it is not acceptable in others. For instance, the text may not be natural language, may be an agglutinating language, or we may simply not want to lose a match when the error involves a separator character. Sequence-retrieving indices are the only answer for these scenarios. However, they are still rather immature and extremely primitive as software systems, with huge space overheads and in many cases poor speedups over sequential searching.

This Part has two chapters. Chapter 8 is devoted to word-retrieving indices. The first part of the chapter analyzes the performance of those indices using some heuristic rules widely accepted in Information Retrieval. We prove that their retrieval times are sublinear in the text size (normally near $O(n^{0.4..0.8})$). We then consider block addressing, showing that it is possible to adjust the block size so that the index remains sublinear in query times and it is at the same time sublinear in space requirements. The band is quite narrow, e.g. we show an example where both quantities grow as $O(n^{0.94})$ for $k = 2$ errors. This analysis holds for exact searching as well, and makes an excellent case of this kind of indices. We confirm our analysis with experimental results.

Chapter 8 also improves the search algorithms currently used in word-retrieving indices. We present a 5-fold improvement to the search in blocks (which applies to extended patterns and regular expressions as well) and an independent 2-fold improvement to search the vocabulary.

Chapter 9 deals with sequence-retrieving indices. We first present a new index based on indexing substrings of the text to implement an off-line version of the fast on-line filter of Section 6.1. The resulting index takes 2 to 4 times the text size and allows reducing the search times, which range from 10% to 60% of the time of the on-line algorithm. This is an index whose space overhead, tolerance to errors and performance shows a good tradeoff, making it a viable alternative in practice. We show that this technique can be implemented over a word-retrieving index, which could allow to solve the problem of allowing errors that involve separators for natural language texts.

The second part of Chapter 9 reconsiders an existing index which uses a suffix tree to simulate on-line traversal avoiding the text redundancies. Our on-line algorithm of Chapter 5 is used instead of dynamic programming. The cost is that we cannot use the smartest techniques to traverse the suffix tree, because they are based on dynamic programming and are not easily bit-parallelizable. We show experimentally, however, that it is much better in practice to use a faster algorithm and a less sophisticated traversal algorithm. On the other hand, pattern partitioning outperforms all the others by far, and we believe that this technique deserves much more study. We propose to simulate it with a $q$-gram index to reduce the space requirements.

Finally, it would be interesting to settle down the question of the possibility of sublinear-time sequence-retrieving indices for natural language text.

# Chapter 8

# Word-Retrieving Indices

In this chapter we address the case of inverted indices used for approximate string matching. As explained in Section 2.10, these indices are able to return only whole words or phrases which match a given search pattern allowing errors. This simplification of the problem is in many cases acceptable, and allows the development of practical indexing schemes. Our contributions are of two kinds: we have analyzed the behavior of these types of indices (finding many unexpected results) and we have proposed and tested some improvements to speed up their existing exponents. This work has been published in [ANZ97, BYN97a, BYN98b, BYN98a].

We remark that our analysis is approximate, since it relies on empirical rules such as Heaps' Law or Zipf's Law (see a complete list of assumptions in Section 2.9), which are only rough approximations to the statistical structure of texts. Moreover, the results are valid only for queries useful to the user (i.e. with reasonable degree of precision). Finally, our analysis considers the average case and gives "big-$O$" (i.e. growth rate) results.

For the experiments of this chapter, we use one of the collections contained in TREC [Har95], namely the Wall Street Journal (WSJ) collection, which contains 278 files of almost 1 Mb each, for a total of 250 Mb of text. To mimic common IR scenarios, all the texts were transformed to lower-case, all separators to single spaces (except line breaks); and stopwords were eliminated. We are left with almost 200 Mb of filtered text. Throughout this chapter we talk in terms of the size of the filtered text, which takes 80% of the original text. To measure the behavior of the index as $n$ grows, we index the first 20 Mb of the collection, then the first 40 Mb, and so on, up to 200 Mb.

## 8.1 Vocabulary Statistics

We present in this section our new results about the statistics of the vocabulary of a text collection. We first show an analytical observation which is independent on the type of queries performed, and later present an empirical result on the amount of expected matching of an approximate query in a vocabulary.

### 8.1.1 Combining Heaps' and Zipf's Laws

Section 2.9 explains the rules which we assume govern the behavior of natural language texts. The most interesting rules are Heaps' Law (which drives the vocabulary growth as the text grows) and Zipf's Law (which rules the frequencies of the vocabulary words in the text). As a reminder, Heaps'

Law states that a text of $n$ words has a vocabulary of size $V = \Theta(n^\beta)$ for $0 < \beta < 1$, and Zipf's Law states that the frequency of the $i$-th most frequent word is $n/(i^\theta H_V^{(\theta)})$, where $H_V^{(\theta)} = \sum_{j=1}^{V} 1/j^\theta$, for some $\theta \geq 1$. For $\theta = 1$ it holds $H_V^{(1)} = \ln V + O(1)$, while for $\theta > 1$ we have $H_V^{(\theta)} = O(1)$.

Something which is not said in the literature is that these two rules can be related. Assume that the least frequent word appears $O(1)$ times in the text (this is more than reasonable in practice, since a large number of words appear only once). Since there are $\Theta(n^\beta)$ different words, then the least frequent word has rank $i = \Theta(n^\beta)$. The number of occurrences of this word is, by Zipf's Law,

$$\frac{n}{i^\theta H_V^{(\theta)}} \quad = \quad \Theta\left(\frac{n}{n^{\beta\theta} H_V^{(\theta)}}\right)$$

and this must be $O(1)$. This implies that, as $n$ grows, $\beta = 1/\theta$. This equality may not hold exactly for real collections. This is because the relation is asymptotical and hence is valid for sufficiently large $n$, and because Heaps' and Zipf's rules are approximations. For instance, in the texts of the TREC collection [Har95], $\beta$ is between 0.4 and 0.6, while $\theta$ is between 1.7 and 2.0. Considering each collection separately, $\beta\theta$ is between 0.80 and 1.04.

### 8.1.2 Vocabulary Matching

An issue which is central to any analysis of the performance of approximate searching on inverted indices is how many words of the vocabulary match a given pattern with $k$ errors. In principle, there is a constant bound to the number of distinct words which match a given pattern with $k$ errors, and therefore we can say that $O(1)$ words in the vocabulary match the pattern. However, not all those words will appear in the vocabulary. Instead, while the vocabulary size increases, the number of matching words that appear increases too, at a lower rate[1]. We show experimentally that a good model for the number of matching words in the vocabulary is $O(n^\nu)$ (with $\nu < \beta$).

For classical word queries we have $\nu = 0$ (i.e. only one word matches). For prefix searching, regular expressions and other multiple-matching queries, we conjecture that the set of matching words grows also as $O(n^\nu)$ if the query is going to be useful in terms of precision. However, this issue deserves a separate study and is out of the scope of this thesis.

Since the average number of occurrences of each word in the text is $n/V = \Theta(n^{1-\beta})$, the average number of occurrences of the pattern in the text is $O(n^{1-\beta+\nu})$. This fact is surprising, since one can think in the process of traversing the text word by word, where each word of the vocabulary has a fixed probability of being the next text word, and hence there is a fixed probability of matching each new text word. Under this model the number of matching words is a fixed proportion of the text size. The fact that this is not the case (demonstrated experimentally in this chapter) shows that this model does not really hold on natural language text.

The root of this fact is not in that a given word does not appear with a fixed probability. Indeed, the Heaps' Law is compatible with a model where each word appears at fixed text intervals. For instance, imagine that Zipf's Law stated that the $i$-th word appeared $n/2^i$ times. Then, the first word could appear in all the odd positions, the second word in all the positions multiple of 4 plus 2, the third word in all the multiples of 8 plus 4, and so on. The real reason for the sublinearity is that, as the text grows, there are more words, and one selects randomly among them. Notice that,

[1]This is the same phenomenon observed in the size of the vocabulary. In theory, the total number of words is finite and therefore $V = O(1)$. But in practice that limit is never reached, and the model $V = O(n^\beta)$ describes reality much better.

asymptotically, this means that the length of the words must be $m = \Omega(\log n)$, and therefore, as the text grows, we search on average longer and longer words. This allows that even in the model where there are $n/\sigma^m$ matches, this number is indeed $o(n)$.

### 8.1.3 Experiments

We present in this section empirical evidence supporting our previous statements. We first measure $V$, the number of words in the vocabulary in terms of $n$ (the text size). Figure 8.1 (left side) shows the growth of the vocabulary. Using least squares we fit the curve $V = 78.81n^{0.40}$. The relative error is very small (0.84%). Therefore, $\beta = 0.4$ for the WSJ collection.

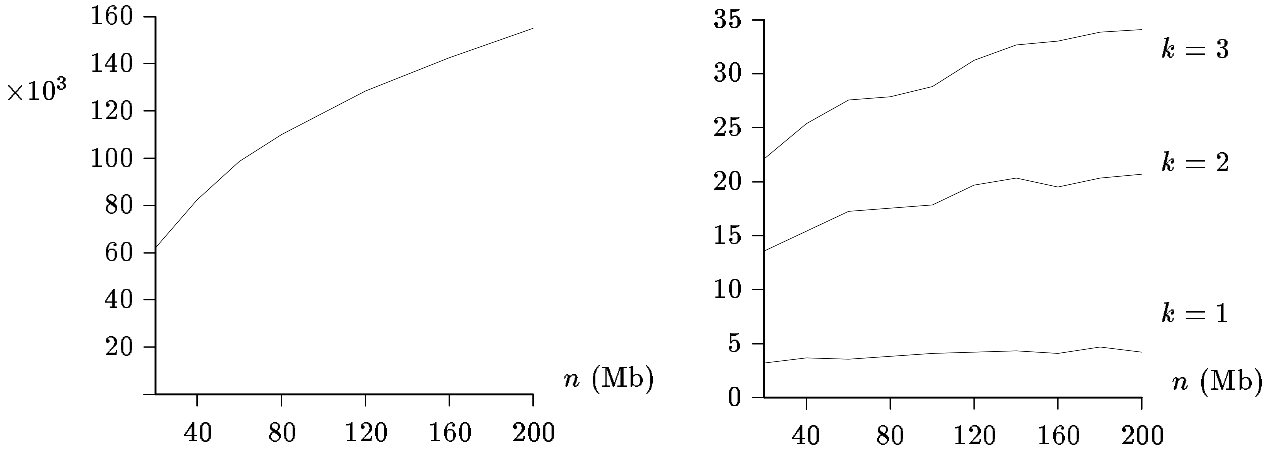


Figure 8.1: Vocabulary tests for the WSJ collection. On the left, the number of words in the vocabulary. On the right, number of matching words in the vocabulary.

We measure now the number of words that match a given pattern in the vocabulary. For each text size, we select words at random from the vocabulary allowing repetitions. This is to mimic common IR scenarios. In fact, not all user queries are found in the vocabulary in practice, which reduces the number of matches. Hence, this test is pessimistic in that sense.

We test $k = 1$, 2 and 3 errors. To avoid taking into account queries with very low precision (e.g. searching a 3-letter word with 2 errors may match too many words), we impose limits on the length of words selected: only words of length 4 or more are searched with one error, length 6 or more with two errors, and 8 or more with three errors.

We perform a number of queries which is large enough to ensure a relative error smaller than 5% with a 95% confidence interval. Figure 8.1 (right side) shows the results. We use least squares to fit the curves $0.31n^{0.14}$ for $k = 1$, $0.61n^{0.18}$ for $k = 2$ and $0.88n^{0.19}$ for $k = 3$. In all cases the relative error of the approximation is under 4%. The exponents are the $\nu$ values mentioned later in this chapter.

We could reduce the variance in the experiments by selecting once the set of queries from the index of the first 20 Mb. However, our experiments have shown that this is not a good policy. The reason is that the first 20 Mb will contain almost all common words, whose occurrence lists grow faster than the average. Most uncommon words will not be included. Therefore, the result is unfair, making the times to look linear when they are in fact sublinear.

## 8.2 Full Inverted Indices

In this section we present the analysis of a full inverted index used for approximate string matching, finding that the retrieval times, even for complex patterns, are sublinear in the text size for reasonable queries.

The *Igrep* software [ANZ97] is a full inverted index described in Section 3.3.1. Although the implementation of this index is not part of this thesis, its analysis is. The aim of this section is to analytically prove that the search times are sublinear in the text size. As an empirical confirmation, we show some experimental results.

Igrep is able to search a single word or an extended pattern that matches a single word, such as a regular expression. It can also search "phrases". A phrase is a sequence of single words or extended patterns. Phrases of $j$ elements match with sequences of $j$ words in the text. Finally, it can search a single element or a phrase allowing up to $k$ errors in the whole match. We analyze now its retrieval times, explaining in the process the algorithms used.

### 8.2.1 Retrieval Times

There are a number of different types of query. Each type involves carrying out different tasks. We first analyze the cost of each task, and then use the results to deduce the cost of each type of query. The description of the tasks follow, together with their analysis. Recall that the size of the vocabulary is $V = O(n^\beta)$, which is normally in the range 0.4 to 0.6 [Hea78, ANZ97].

**bin-search:** binary searching a word in the vocabulary and retrieving the list. Since the search is binary, we have $O(\log n^\beta) = O(\log n)$ cost for this type of task.

**seq-search:** sequentially searching a word in the vocabulary is $O(n^\beta)$. This is the case of regular expressions, extended patterns and approximate string matching, since all them can be reduced to deterministic automata. Alternative schemes achieve $O(kn^\beta)$ time for approximate searching or $O(mn^\beta/w)$ for regular expressions and extended patterns. These quantities can all be considered $O(n^\beta)$ for our purposes.

**lst-merge:** list merging of $j$ lists of occurrences happens in approximate searching, extended patterns, etc. Since the average size of each list of occurrences is $n/V = O(n^{1-\beta})$ and we merge ordered lists to produce an ordered list, we work $O(n^{1-\beta} j \log j)$ (using a heap of size $j$).

We point out now the times for each type of query, as follows:

**Simple words:** the word is searched in the vocabulary and the list of occurrences is retrieved. If the search is binary, we have $O(\log n^\beta) = O(\log n)$ search cost for this type of query. With other data structures the search time can be even less, e.g. $O(m)$ by using hashing or a trie.

**Phrases of $j$ simple words:** each one of the $j$ words is searched in the vocabulary at a total cost of $O(j \log n)$. Then, the $j$ lists of occurrences are "intersected", in the sense that we check the text positions where the $j$ words are contiguous and in the proper order (the algorithm is very similar to a list intersection). To perform this pseudo-intersection, the shortest among the $j$ lists is selected, and its positions are binary searched inside the other lists to verify its surrounding area. We prove now that the shortest list among $j$ is $O(1)$ length and therefore the total cost of the list intersection is $O(j \log n)$.

To show that, we use the Zipf's Law and our result of Section 8.1.1, namely $\beta\theta = 1$. If we consider $X_1..X_j$ the rank of the words present in a phrase (which are uniformly distributed over $[1..V]$), and call $f(i)$ the number of occurrences of the $i$-th most frequent word, we have

$$P(\min f(X_i) \ \geq a) = (P(f(X_1) \geq a))^j$$

and since $f(i) = n/(i^\theta H_V^{(\theta)})$, we invert $f$ to get

$$\left( P \left( X_1 \leq \left( \frac{n}{a\ H_V^{(\theta)}} \right)^{1/\theta} \right) \right)^j$$

which given that $X_i$ distributes uniformly over $[1..V]$ is

$$\left( \left( \frac{n}{a\ H_V^{(\theta)}} \right)^{1/\theta} \frac{1}{V} \right)^j \quad = \quad (a\ H_V^{(\theta)})^{-j/\theta}$$

where the last equality is obtained by cancelling $n^{1/\theta}$ with $1/V = 1/n^\beta = 1/n^{1/\theta}$.

Hence, the expectation of the length of the shortest list is

$$\sum_{a=1}^{V} P(\min f(X_i) \ \geq a) \quad \leq \quad \frac{1}{\left(H_V^{(\theta)}\right)^{j/\theta}} \sum_{a \geq 1} \frac{1}{a^{j/\theta}}$$

which is $O(1)$ for $j > \theta$. This is typically out of question for phrases of three words or more. However, for $j = 2$ that may not be the case, although it is generally true (e.g. in the TREC collection, where $\theta < 2$ always holds). If $\theta \geq 2$, we can bound the summation with an integral to get that the expectation is smaller than $1 + V^{1-j/\theta}/(j-1) + O(1/j) = O(n^{\beta(1-\beta j)})$. In that case the total cost of the intersection is $O(n^{\beta(1-\beta j)} \log n)$.

**Extended patterns, regular expressions and approximate word matching:** the pattern is sequentially searched in the vocabulary and all the lists of occurrences of the resulting matches are merged. The time of the sequential search is $O(n^\beta)$. If $p$ vocabulary words match, we merge them using a heap. The lists are of average length $n/V = n^{1-\beta}$, and therefore the total merge time is $O(pn^{1-\beta} \log p)$. Hence, the total cost of this type of query is $O(n^\beta + pn^{1-\beta} \log p)$.

**Phrases formed with complex patterns:** we perform the $j$ searches as those for the above expressions (which add up $O(j(n^\beta + pn^{1-\beta} \log p))$), plus an intersection of the lists. Those lists do not obey the Zipf's Law because they come from the union of random words, and are of average length $O(pn^{1-\beta})$. Hence, the trick of using the shortest list does not work and the total intersection cost for the $j$ lists (using a heap again) is $O(pn^{1-\beta} j \log j)$. The total time is thus $O(j(n^\beta + pn^{1-\beta} \log(pj)))$.

**Approximate phrase matching of $j$ words and $k$ errors:** is solved by searching each phrase element with up to $k$ errors. For each of the $j$ elements, $k+1$ lists are initialized where the words that matched each element with each number of errors are stored. This costs $j$ **seq-search**es considering a number of different **lst-inters**, which has a cost of $O(j(n^\beta + pn^{1-\beta} \log p) + \binom{k+j}{j} pn^{1-\beta} j \log j)$, under the same conditions as above. This is because we try all alternatives of selecting a list of each word, such that the total number of errors does not exceed $k$. That is

$$\sum_{i=0}^{k} \binom{i+j-1}{i} = \binom{k+j}{j}$$

Again, the lists of occurrences are $O(pn^{1-\beta})$ size, where $p$ corresponds to searching a single word with $k$ errors.

In all these results, we can consider $j$ and $k$ as small constants. On the other hand, as we show in Sections 8.1.2 and 8.1.3, $p = O(n^{\nu})$ for $\nu$ in the range 0.1 .. 0.2 if the precision is reasonably good (i.e. $k \leq 3$ for words). The same should happen for complex patterns, since otherwise a large $p$ value means a query which has too low precision and is of no use to the final user. This condition can be detected beforehand.

Therefore, exact search queries can be solved at $O(\log n)$ expected time, while complex patterns or searching allowing errors costs $O(n^{\max(\beta,1-\beta+\nu)})$, which is in the range $O(n^{0.4..0.8})$ depending on the vocabulary size and the complexity of the search. In reasonable cases it is $O(n^{0.6})$, which is near $O(\sqrt{n})$. We also point out that the disk accesses to the index are sequential (except for buffering limitations). This matches with the results of Section 8.3.

### 8.2.2 Experimental Results

In this section we present some experimental results which are useful to confirm the analysis[2]. These results were obtained using an isolated Sun SparcStation 4 with 128 megabytes of RAM running Solaris 2.5.1. The text used was part of a newer TREC collection (TREC-3), where the ZIFF collection has near 700 Mb. The experiments use this collection. More details and experiments are found in the original paper [ANZ97, ANZ98], where the Heaps' and Zipf's Laws are experimentally validated, and other parameters such as index construction time and space are studied. In this section we only present the results which are relevant to our analysis.

First, we experimentally validate the fact that the length of the shortest list is constant if selected among 3 or more random lists. Figure 8.2 shows the results. As it can be seen, the shortest among two lists remains somewhat increasing, while among more than two lists stabilize in the long term. This is especially clear for $j = 5$, while the same should happen to the others for longer texts.

We now consider time complexity. The experiments to measure query times considered exact and approximate queries ($k = 0, 1, 2, 3$), phrase patterns containing 1, 2, 3, 4, and 5 words. The patterns were randomly chosen from the texts, avoiding patterns containing stopwords.

Figure 8.3 shows the retrieval times as a function of the number of words $j$ in the query. As analytically predicted, the costs for zero errors are basically independent on the number of words, with a very slow increment for the merging process (where the qualifying words are binary searched in the other lists). For one error or more, the costs increase linearly with the number of words, as the analysis predicts. This is because each of the $j$ lists comes from merging many short lists, and then we have to intersect those $j$ lists.

What the analysis did not predict is the sharp increase for $j = 2$. This does not come from generating the lists for each pattern, but from the intersections. For instance, for $k = 2$ we have a higher cost for $j = 2$, which comes from intersecting the first pattern with one error and the second

[2]The results come from a joint work with Nivio Ziviani and Marcio Drumond Araújo, which is not yet published [ANZ98]. The implementation of the index is part of Drumond's Master's Thesis and not of ours.

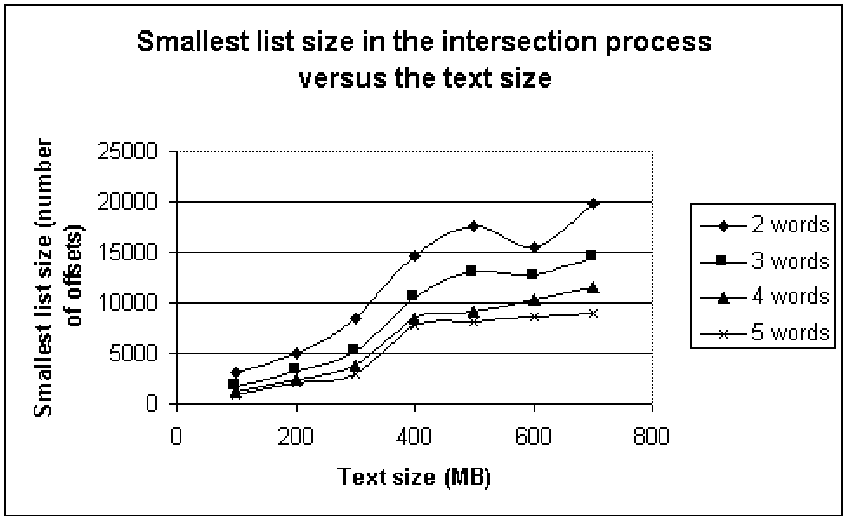


Figure 8.2: Length of the shortest among $j$ lists, for $j = 2$ to 5.

pattern with one error (which we call the combination [1,1]). The other alternatives (where one word matches exactly and the others with two errors, i.e. [0,2] and [2,0]) are cheaper since one of the lists is typically very short. When we move to $j = 3$ then any possible combination with $k = 2$ errors contains one word that matches exactly, and hence all the merges are cheaper. The same happens to $k = 3$, where with $j = 2$ we have expensive combinations such as [1,2] and [2,1], and with $j = 3$ we have [1,1,1], but the costs are cheaper for $j \geq 4$. This shows empirically that the costs are high for $1 < j \leq k$.

We show in Figure 8.4 the retrieval times as a function of the number $k$ of errors allowed. As it can be seen, the retrieval times increase sharply as the number of errors goes from 0 to 3. For more than 3 errors the amount of text retrieved makes the query useless for any practical purpose.

Finally, we show that the retrieval times are, as predicted, sublinear in the text size if the number of allowed errors is not too high. Figure 8.5 shows the performance as a function of the text size, for queries of one to three words. Table 8.1 shows the results of the least squares fitting of the curves (up to five words, although we do not plot the case of four and five words here). We have used the model $O(n^x)$ for all the entries, even for $k = 0$, since this includes the time to read the results from disk. The relative error of the approximation goes from 1% to 9%. As it can be seen, the results are as predicted in the sense that sublinear times are obtained when the number of allowed errors is reasonable. Phrase patterns are also affected by the phenomenon described before, which worsens the complexities for $1 < j \leq k$.

| | 1 word | 2 words | 3 words | 4 words | 5 words |
|---|---|---|---|---|---|
| $k = 0$ | $.064n^{.531}$ | $.057n^{.572}$ | $.054n^{.582}$ | $.050n^{.595}$ | $.049n^{.601}$ |
| $k = 1$ | $.087n^{.524}$ | $.043n^{.742}$ | $.059n^{.698}$ | $.066n^{.700}$ | $.078n^{.691}$ |
| $k = 2$ | $.076n^{.589}$ | $.028n^{.969}$ | $.053n^{.836}$ | $.075n^{.786}$ | $.010n^{.754}$ |
| $k = 3$ | $.058n^{.698}$ | $.021n^{1.239}$ | $.033n^{1.089}$ | $0.60n^{.968}$ | $.082n^{.923}$ |

Table 8.1: Least squares fitting for the retrieval times.

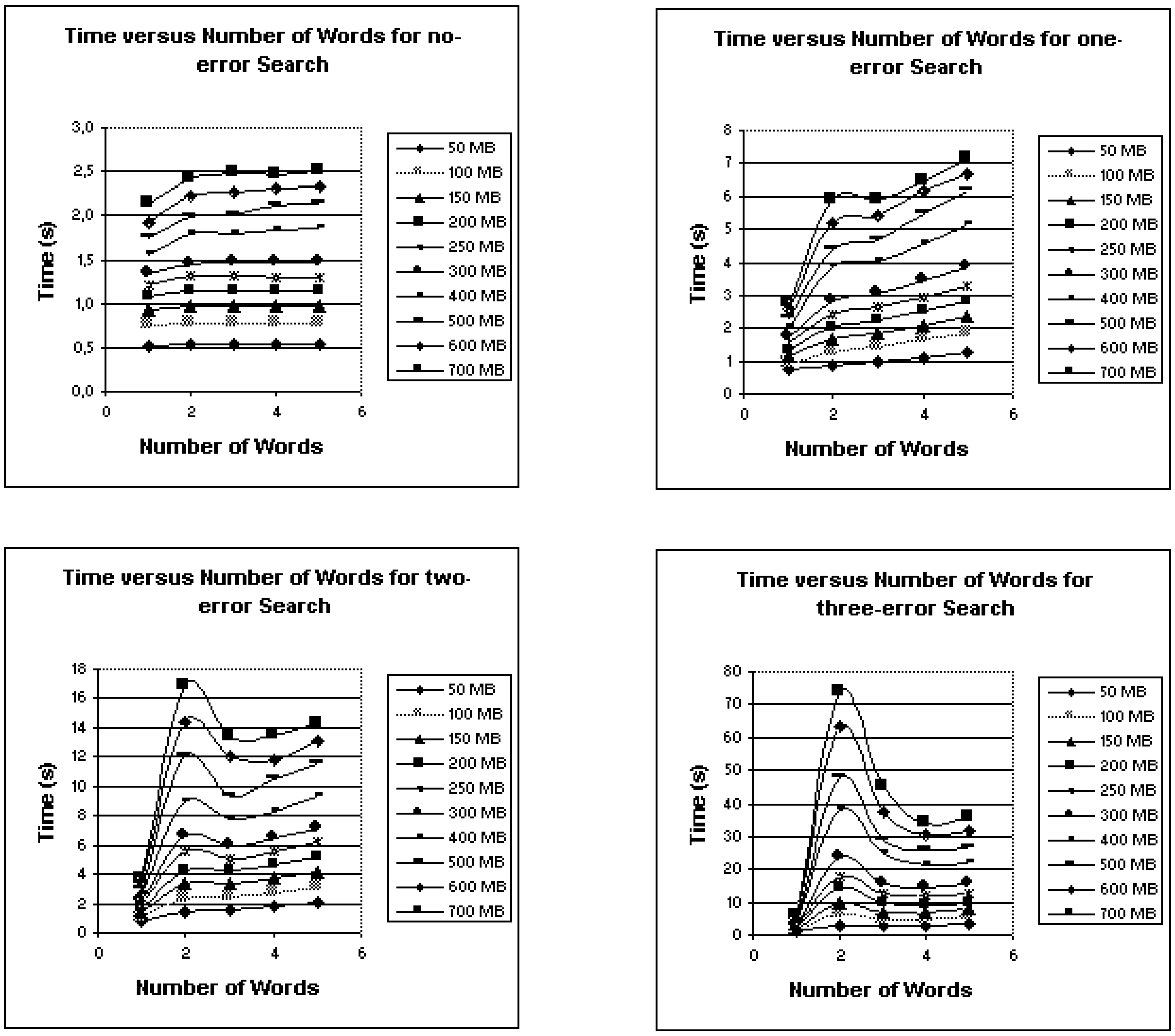
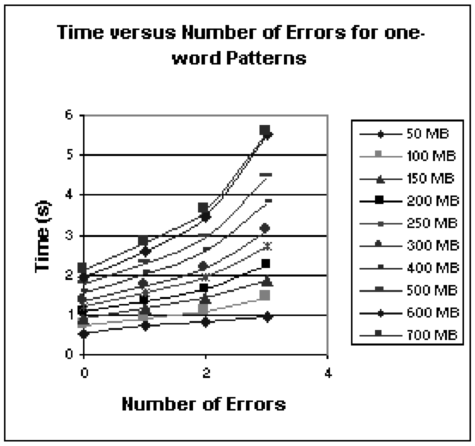
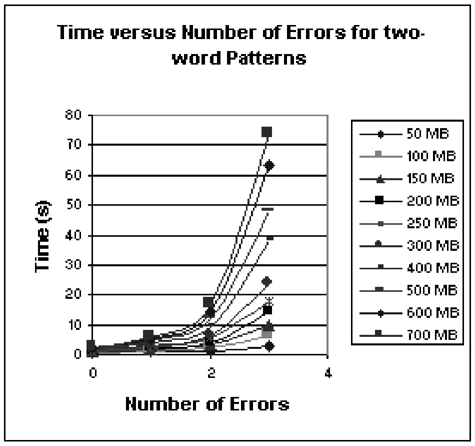
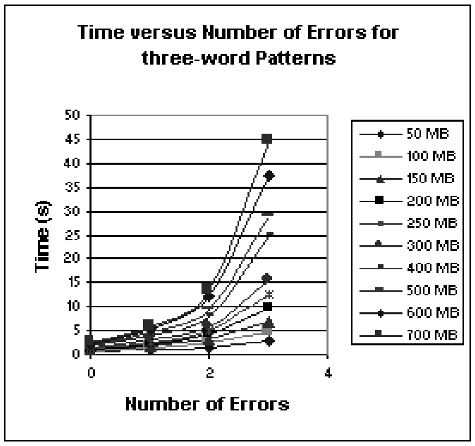
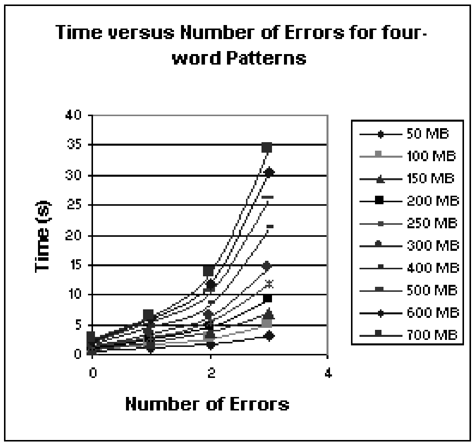


Figure 8.3: Retrieval times as a function of the number of words in the query.









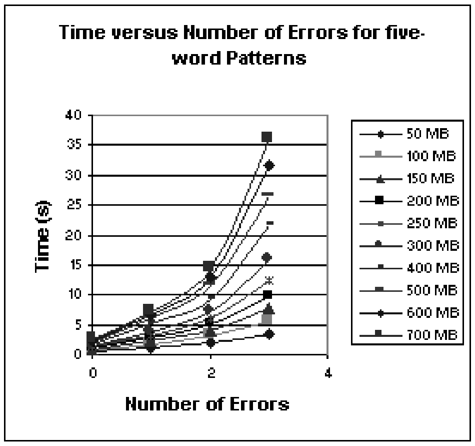


Figure 8.4: Retrieval times as a function of the number of errors allowed.

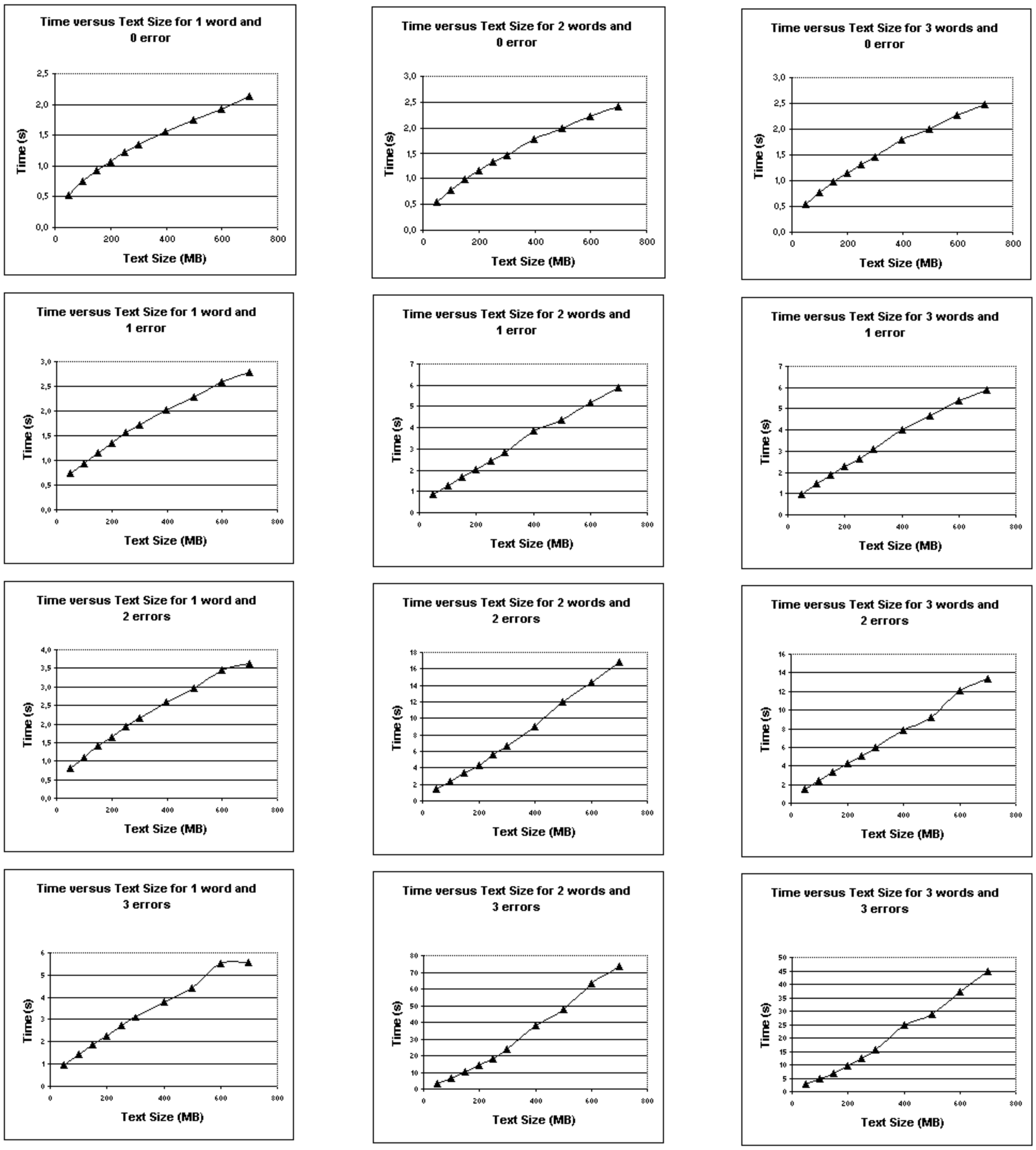


Figure 8.5: Retrieval times for patterns of one to three words, as a function of the text size.

### 8.2.3 Differential Pointers

In the previous analysis we assume that the pointers are not compressed. This allows performing a binary search on the list of pointers, therefore avoiding the cost of a sequential search. A very simple technique to reduce the space requirements of inverted files is to store *differential* pointers, i.e. each pointer gives the difference between its real value and the previous one. This works well because the list of pointers of each vocabulary word has increasing values. In practice this allows reducing the space in 50% or more. However, we cannot perform a binary search anymore (except by storing absolute pointers at regular intervals). If we have to sequentially process the lists, the analysis of the retrieval times is identical except for the phrases of simple words, which now must be processed by merging rather than by binary searching. That is, we pay $O(n^{1-\beta} j \log j)$ time for this case. In particular, if we have to deliver the resulting text positions, then even searching one simple word costs $O(n^{1-\beta})$, which is its expected number of occurrences in the text.

## 8.3 Block Addressing Inverted Indices

*Glimpse* and *Igrep* are two extremes of a more general idea (see Section 3.3.1). The first one (in its "tiny" index) cuts the text in a fixed number of blocks ("block addressing") to reduce space requirements, and sequentially verifies the occurrences inside the matching blocks. However, when the text is large enough all the blocks have to be checked and the index is not useful anymore. *Igrep* stores all the positions of all the words, which makes it resistant to huge text sizes but imposes a linear space overhead. *Glimpse* has sublinear space overhead, while *Igrep*, as shown in the previous section, has sublinear query time.

In this section we study the use of block addressing to obtain indices which are sublinear in space *and* in query time, and show analytically a range of valid combinations to achieve this. The combined sublinearity means that, as the text grows, the space overhead of the index and the time to answer a query become less and less significant as a proportion of the text size. We validate this analysis with extensive experiments, obtaining typical performance figures.

Once we establish our results for block addressing indices where the block size is fixed, we consider an interesting particular case of document addressing (where the documents are of variable size): we use recently obtained statistics from the distribution of the page sizes in the Web [CB96] and apply our machinery to determine the space overhead and retrieval time of an index for a collection of Web pages. We show that having documents of different sizes reduces space requirements in the index but increases search times if the documents have to be traversed.

We recall from Section 2.10 the index structure: the text is logically divided into "blocks". The index stores all the different words of the text (the vocabulary). For each word, the list of the blocks where the word appears is kept. We call $b$ the size of the blocks and $r$ the number of blocks, so that $n \approx rb$ (see Figure 8.6).

At this point the reader may wonder which is the advantage of pointing to artificial blocks instead of pointing to documents (or files), this way following the natural divisions of the text collection. This issue is raised in Section 2.9, but we explain it more in detail here. If we consider the case of simple queries (say, one word), where we are required to return only the list of matching documents, then pointing to documents is a very adequate choice. Moreover, as we see later, it may reduce space requirements with respect to using blocks of the same size. Moreover, if we pack many short documents in a logical block, we will have to traverse the matching blocks (even for these simple queries) to determine which documents inside the block actually matched.

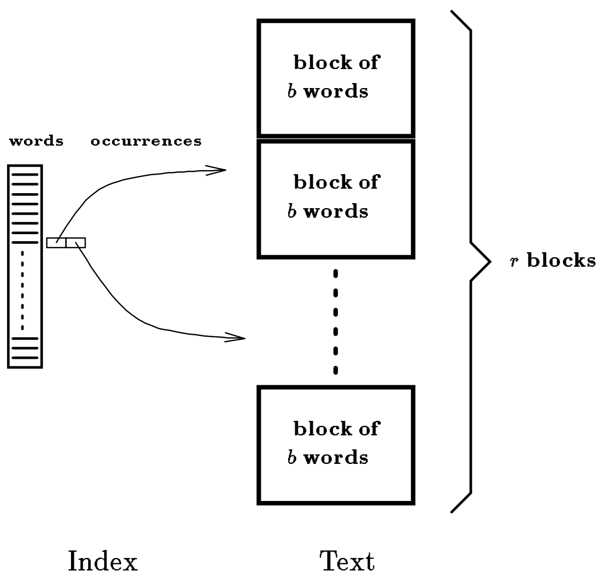


Figure 8.6: The block-addressing indexing scheme.

However, consider the case where we are required to deliver the exact positions which match a pattern. In this case we need to sequentially traverse the matching blocks or documents to find the exact positions. Moreover, in some types of queries such as phrases or proximity queries, the index can only tell that two words are in the same block, and we need to traverse it in order to determine if they form a phrase.

In this case, pointing to documents of different sizes is not a good idea because larger documents are searched with higher probability and searching them costs more. In fact, the expected cost of the search is directly related to the variance in the size of the pointed documents. This suggests that if the documents have different sizes it may be a good idea to (logically) partition large documents into blocks and to put together small documents, such that blocks of the same size are used.

### 8.3.1 Average Space-Time Trade-offs

*Glimpse* and *Igrep* are two extremes of a single idea. *Glimpse* achieves small space overhead at the cost of sequentially traversing parts of the text. *Igrep* achieves better performance by maintaining a large index. We study in this section the possibility of having an intermediate index, which is sublinear in size and query performance at the same time. We show that this is possible in general, under reasonable assumptions.

In Section 8.2 we considered in detail the case of complex queries formed by phrases and extended patterns. Those queries in most cases introduced additional factors to the complexity of the basic search, which for the sake of sublinearity with respect to $n$ can be disregarded. Therefore, we simply consider the case of approximate word searching from now on, knowing that the result is generalizable to a more complex query.

#### 8.3.1.1 Query Time Complexity

To search an approximate pattern, a first pass runs an on-line algorithm over the vocabulary. The sets of blocks where each matching word appears are collected. For each such block, a sequential search is performed on that block.

The sequential pass over the vocabulary is linear in $V$, hence it is $\Theta(n^\beta)$, which is sublinear in the size of the text. On the other hand, the blocks to work on in the text are those including some (exact or approximate) occurrence of the pattern. We model the process as follows: an approximate search first selects $O(n^\nu)$ random words from the vocabulary, which is of size $\Theta(n^\beta)$. Hence, the probability of a given vocabulary word to be selected by the search is $O(n^{\nu-\beta})$. To determine whether a block needs to be searched or not, we take each one of its $b$ words and look if they are selected in the vocabulary. We work on the block if any of its words has been selected in the vocabulary.

The probability of a word from the block to be selected is $O(n^{\nu-\beta})$. The probability that none of the words in the block is selected is therefore $(1 - O(n^{\nu-\beta}))^b$. The total amount of work is obtained by multiplying the number of blocks ($r$) times the work to do per selected block ($b$) times the probability that some word in the block is selected. This is

$$\Theta\left(rb\left(1-\left(1-n^{\nu-\beta}\right)^b\right)\right) \quad = \quad \Theta\left(n\left(1-e^{-b/n^{\beta-\nu}}\right)\right) \tag{8.1}$$

where for the last step we used that $(1-x)^y = e^{y\ln(1-x)} = e^{y(-x+O(x^2))} = \Theta(e^{-yx})$ provided $x = o(1)$.

We are interested in determining in which cases the above formula is sublinear in $n$ or not. Expressions of the form "$1 - e^{-x}$" appear frequently in this analysis. We observe that they are $O(x)$ whenever $x = o(1)$ (since $e^{-x} = 1 - x + O(x^2)$). On the other hand, if $x = \Omega(1)$, then $e^{-x}$ is far away from 1, and therefore "$1 - e^{-x}$" is $\Omega(1)$.

For the search cost to be sublinear, it is thus necessary that $b = o(n^{\beta-\nu})$, which we call the "condition for time sublinearity". When this condition holds, we derive from Eq. (8.1) that

$$Time \quad = \quad \Theta\left(n^\beta \; + \; bn^{1-\beta+\nu}\right) \tag{8.2}$$

which matches the results of Section 8.2, where $b = 1$.

#### 8.3.1.2 Space Complexity

We consider space now. The average size of the vocabulary itself is already sublinear. However, the total number of references to blocks where each word appears may be linear (it is truly linear in the case of full inversion, which corresponds to single-word blocks, i.e. $b = 1$ as studied in Section 8.2).

The analysis is very simple if we notice that each block of size $b$ has $O(b^\beta)$ different words, by Heaps' Law. Each different word that appears in each different block will correspond to a different entry in the inverted index. Hence, the size of this index is just the number of different words of each block times the number of different blocks, that is,

$$Space \quad = \quad \Theta\left(rb^\beta\right) \quad = \quad \Theta\left(rb^{1/\theta}\right) \tag{8.3}$$

Hence, for the space to be sublinear we just need $r = o(n)$, or equivalently, $b = \omega(1)$.

However, we have assumed the validity of an asymptotic rule such as Heaps' Law for blocks, which are much smaller than the whole text. Figure 8.7 shows the evolution of the $\beta$ value as the text collection grows. We show its value for up to 1 Mb. As it can be seen, $\beta$ starts at a higher value and converges to the definitive 0.40 as the text grows. For 1 Mb it has almost reached its definitive value. Hence, the Heaps' Law holds for smaller blocks but the $\beta$ value is higher than its asymptotic limit.

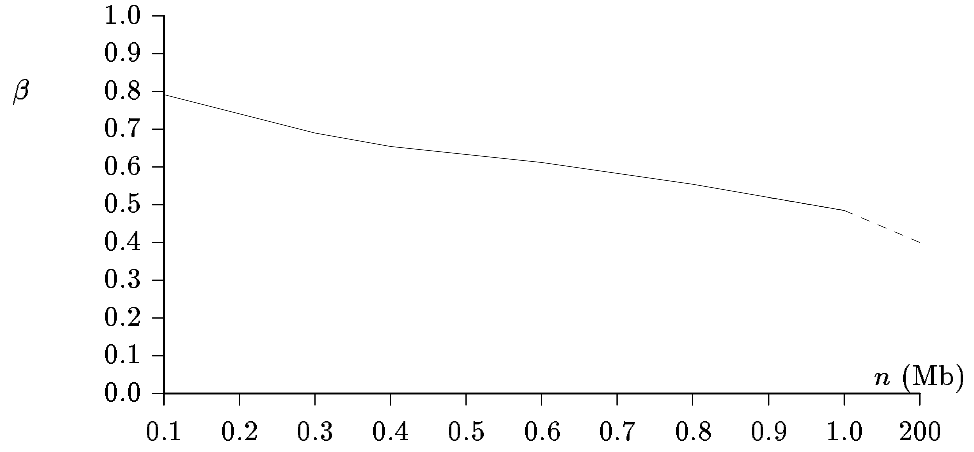


Figure 8.7: Value of $\beta$ as the text grows. We added at the end the value for the 200 Mb collection.

We analyze now the space usage using Zipf's Law instead of Heaps' Law. The analysis is more complex in this case, but the result is the same if we replace $\beta$ by $1/\theta$. This confirms that for both rules to be valid it must hold $\beta = 1/\theta$. In the analysis that follows, $\beta$ and $1/\theta$ can be used interchangeably whenever the space complexity is involved. In particular, we use $1/\theta$ to draw later the actual numbers, since it is more precise.

Suppose that a word appears $\ell$ times in the text. The same argument used for Eq. (8.1) shows that it appears in $\Theta(r(1-e^{-\ell/r}))$ blocks on average. Recall that the index stores an entry in the list of occurrences for each different block where a word appears. Under the Zipf's Law, the number of occurrences of the $i$-th most frequent word is $\ell_i = n/(i^\theta H_V^{(\theta)})$. Therefore, the number of blocks where it appears is

$$\Theta\left(r\left(1-e^{-\ell_i/r}\right)\right) = \Theta\left(r\left(1-e^{-b/(i^\theta H_V^{(\theta)})}\right)\right)$$

and the total number of references to blocks is

$$r \sum_{i=1}^{V} 1 - e^{-b/(i^\theta H_V^{(\theta)})} \tag{8.4}$$

a summation which is hard to solve exactly. However, we can still obtain the required big-$O$ information. We show now that there is a threshold $a$ such that

$$a = \left(\frac{b}{H_V^{(\theta)}}\right)^{1/\theta}$$

1. The $O(a)$ most frequent words appear in $\Theta(r)$ blocks, and therefore contribute $\Theta(ar)$ to the size of the lists of occurrences. This is because each term of the summation (8.4) is $\Omega(1)$ provided $b = \Omega\left(i^\theta H_V^{(\theta)}\right)$ which is equivalent to $i = O(a)$.

2. The $O(V-a)$ least frequent words appear nearly each one in a different block, that is, if the word appears $\ell$ times in the text, it appears in $\Omega(\ell)$ blocks. This is because $r(1-e^{-\ell/r}) = \Theta(\ell)$ whenever $\ell = o(r)$. For $\ell_i = n/(i^\theta H_V^{(\theta)})$, this is equivalent to $i = \omega(a)$.

   Summing the contributions of those lists and bounding with an integral we have

$$\begin{aligned}\sum_{i=a+1}^{V} \frac{n}{i^\theta H_V^{(\theta)}} &= \frac{n}{H_V^{(\theta)}} \frac{1/a^{\theta-1} - 1/V^{\theta-1}}{\theta-1} (1+o(1)) \\ &= \Theta\left(\frac{n}{a^{\theta-1}}\right) = \Theta(ar)\end{aligned}$$

   where we realistically assume $\theta > 1$ (we consider the case $\theta = 1$ shortly).

Therefore, the total space for the lists of occurrences is always $\Theta(ar) = \Theta(rb^{1/\theta})$ for $\theta > 1$.

We have left aside the case $\theta = 1$, because it is usually not true un practice. However, we show now what happens in this case. We have that $a = \Theta(b/\log V) = \Theta(b/\log n)$. Summing the two parts of the vocabulary we have that the space for the lists of occurrences is

$$\Theta\left(\frac{n}{\log n} + n\left(1 - \frac{\log b}{\log n} + \frac{\log\log n}{\log n}\right)\right)$$

which is sublinear provided $b = \Omega(n^\delta)$, for every $\delta < 1$ (e.g. $b = n/\log n$). This condition opposes to the one for time sublinearity, even for classical searches with $\nu = 0$. Therefore, it is not possible to achieve combined sublinearity in this (unrealistic) case.

#### 8.3.1.3 Combined Sublinearity

Simultaneous time and space complexity can be achieved whenever $b = o(n^{\beta-\nu})$ and $r = o(n)$. To be more precise, assume we want to spend

$$Space = \Theta(n^\gamma)$$

space for the index. Given that the vocabulary alone is $O(n^\beta)$, $\gamma \geq \beta$ must hold. Solving $rb^\beta = n^\gamma$ with Eq. (8.3) we have

$$r = \Theta\left(n^{\frac{\gamma-\beta}{1-\beta}}\right) \quad , \quad b = \Theta\left(n^{\frac{1-\gamma}{1-\beta}}\right)$$

Since the condition for time sublinearity imposes $b = o(n^{\beta-\nu})$, we conclude

$$\gamma > \mu = 1 - (1-\beta)(\beta-\nu)$$

(which implies $\gamma \geq \beta$). In that case, the time complexity (computed using Eq. (8.2)) becomes

$$Time = \Theta\left(n^\beta + n^{1-\beta+\nu+\frac{1-\gamma}{1-\beta}}\right)$$

(and $\Theta(n)$ if $\gamma \leq \mu$). Note that the above expression turns out to be just the number of matching words in the text times the block size.

The combined $Time \times Space$ complexity is

$$Time \times Space = \Theta\left(n^{\beta+\gamma} + n^{1-\beta+\nu+\frac{1-\gamma\beta}{1-\beta}}\right)$$

which, if $2\beta \leq 1+\nu$, is minimized for $\gamma = 1$ (full inversion), where $Space = \Theta(n)$ and $Time = \Theta(n^{\max(\beta,1-\beta+\nu)})$, as obtained in Section 8.2. On the other extreme, the "tiny" index of *Glimpse* corresponds to $\gamma = \beta$, in which case $Time = \Theta(n)$ and $Space = \Theta(n^\beta)$, i.e. just the necessary to store the vocabulary. On the other hand, when $2\beta > 1+\nu$, the minimum is obtained for $\gamma = 1-(2\beta-\nu)(1-\beta)/(2-\beta)$.

The practical values of the TREC collection show that $\gamma$ must be larger than 0.77 .. 0.89 in practice, in order to answer queries with at least one error in sublinear time and space. Figure 8.8 shows possible time and space combinations for $\beta = 0.4$ and $\theta = 1.87$, values that hold for the WSJ collection we use in the experiments. The values correspond to searching with $k = 2$ errors, which, as found in Section 8.1.3, has $\nu = 0.18$. If less space is used, the time remains linear (as in *Glimpse*).

The figure also shows schematically the valid time and space combinations. We plot the exponents of $n$ for varying $\gamma$. As the plot shows, the only possible combined sublinear complexity is achieved in the range $\mu < \gamma < 1$, which is quite narrow.

| Space | Time |
|---|---|
| $n^{0.90}$ | $n^{0.99}$ |
| $n^{0.92}$ | $n^{0.95}$ |
| $n^{0.94}$ | $n^{0.91}$ |
| $n^{0.96}$ | $n^{0.87}$ |
| $n^{0.98}$ | $n^{0.82}$ |
| $n^{1.00}$ | $n^{0.78}$ |

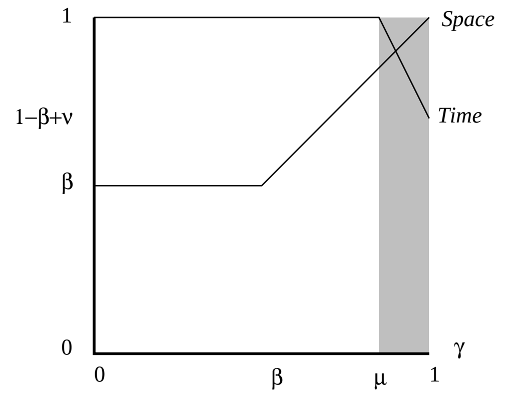


Figure 8.8: On the left, valid combinations for time and space complexity assuming $\theta = 1.87$, $\beta = 0.4$ and $\nu = 0.18$. On the right, time and space complexity exponents. The area of combined sublinearity is shaded.

We end this section with a couple of practical considerations regarding this kind of index. First, using blocks of fixed size imposes no penalty on the overall system, since the block mechanism is a logical layer and the files do not need to be physically split or concatenated.

Another consideration that arises is how to build the index incrementally if the block size $b$ has to vary when $n$ grows. Reindexing each time with a new block size is impractical. A possible solution is to keep the current block size until it should be doubled, and then process the lists of occurrences making equal all blocks numbered $2i$ with those numbered $2i+1$ (and deleting the resulting duplicates). This is equivalent to deleting the least significant bit of the block numbers. The process is linear in the size of the index (i.e. sublinear in the text size) and fast in practice. Splitting blocks due to deletions in the text collection is however more complicated, but many collections never decrease significantly in size.

### 8.3.2 Analyzing the Web

In [CB96], an empirical model for the distribution of the sizes of the Web pages is presented, backed by thorough experiments. This distribution is as follows: the probability that a Web page is of size

$x$ is

$$p(x) = \frac{\lambda k^{\lambda}}{x^{1+\lambda}}$$

for $x \geq k$, and zero otherwise. The cumulative distribution is

$$F(x) = 1 - \left(\frac{k}{x}\right)^{\lambda}$$

where $k$ and $\lambda$ are constants dependent on the particular collection. $k$ is the minimum document size, and $\lambda = 1.36$ when considering textual data.

As explained before, pointing to documents instead of blocks may or may not be convenient in terms of query times. We analyze now the space and later the time requirements when we point to Web pages.

As the Heaps' Law states that a document with $x$ words has $x^{\beta}$ different words, we have that each new document of size $x$ added to the collection will insert $x^{\beta}$ new references to the lists of occurrences (since each different word of each different document has an entry in the index). Hence the average number of new entries in the occurrence list per document is

$$\int_k^{\infty} p(x) x^{\beta} dx = \frac{\lambda k^{\beta}}{\lambda - \beta} \qquad (8.5)$$

To determine the total size of the collection, we consider that $r$ documents exist, whose average length is

$$b^* = \int_k^{\infty} p(x) x dx = \frac{\lambda k}{\lambda - 1} \qquad (8.6)$$

and therefore the total size of the collection is

$$n = \frac{r\lambda k}{\lambda - 1} \qquad (8.7)$$

The size of the vocabulary in the final collection is

$$n^{\beta} = \left(\frac{r\lambda k}{\lambda - 1}\right)^{\beta}$$

and the final size of the occurrence list is (using Eqs. (8.5) and then (8.7))

$$\frac{r\lambda k^{\beta}}{\lambda - \beta} = \frac{\lambda - 1}{\lambda - \beta} \frac{1}{k^{1-\beta}} n \qquad (8.8)$$

A first result is that the space of the index is $\Theta(n)$ (this should be clear since $b^* = O(1)$). We consider now what happens if we take the average document length and use blocks of that fixed size (splitting long documents and putting short documents together as explained). In this case, the size of the vocabulary is $O(n^{\beta})$ as before, and we assume that each block is of a fixed size $b = zb^* = z\lambda k/(\lambda - 1)$ (Eq. (8.6)). We have introduced a constant $z$ to control the size of our blocks. In particular, if we use the same number of blocks as Web pages, then $z = 1$. Then the size of the lists of occurrences is

$$(r/z) b^{\beta} = r \frac{\lambda^{\beta} k^{\beta} z^{\beta}}{z(\lambda - 1)^{\beta}} = \left(\frac{\lambda - 1}{z\lambda k}\right)^{1-\beta} n$$

(using Eq. (8.7) for the last step). Now, if we divide the space taken by the index of documents by the space taken by the index of blocks (using the previous equation and Eq. (8.8)), the ratio is

$$\frac{\text{document index}}{\text{block index}} = \frac{z^{1-\beta}\lambda^{1-\beta}(\lambda-1)^{\beta}}{\lambda-\beta} \tag{8.9}$$

which is independent on $k$ and rounds 80% for $z = 1$ and $\beta = 0.4..0.6$. This shows that indexing documents yields an index which takes 80% of the space of a block addressing index, if we have as many blocks as documents. Figure 8.9 shows the ratio as a function of $\lambda$ and $\beta$. As it can be seen, the result varies slowly with $\beta$, while it depends more on $\lambda$ (tending to 1 as the document size distribution is more uniform).

The fact that the ratio varies so slowly with $\beta$ is good because we already know that the $\beta$ value is quite different for small documents. As a curiosity, notice that there is a $\beta$ value which gives the minimum ratio for document versus block index (i.e. the worst behavior for the block index). This is

$$\beta^* = \lambda - \frac{1}{\ln \frac{z\lambda}{\lambda-1}}$$

which is $\beta^* \approx 0.61$ for $z = 1$.

If we want to have the same space overhead for the document and the block indices, we simply make the expression of Eq. (8.9) equal to 1 and obtain $z \approx 1.4..1.7$ for $\beta = 0.4..0.6$, i.e. we need to make the blocks larger than the average of the Web pages. This translates into worse search times. By paying more at search time we can obtain smaller indices (letting $z$ grow over 1.7).

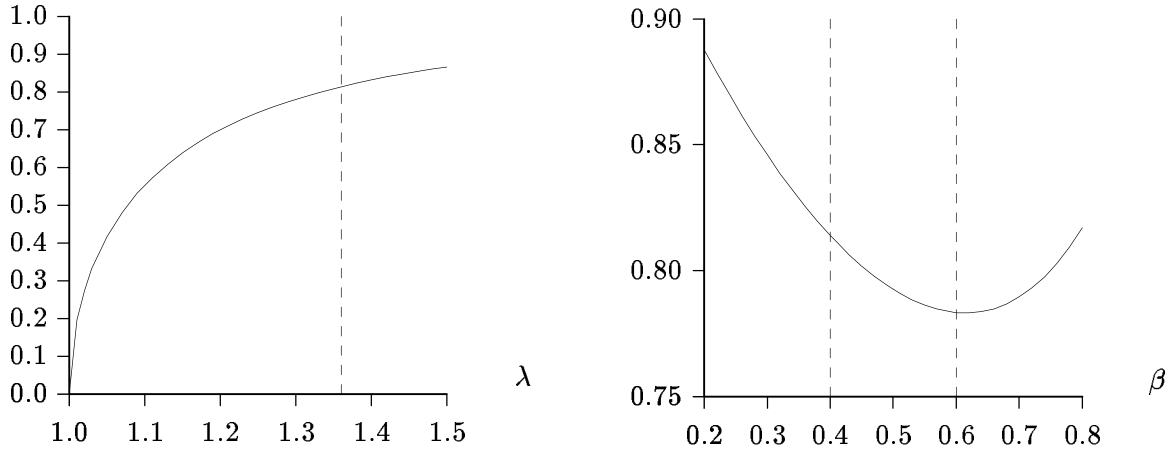


Figure 8.9: On the left, ratio between both indices as a function of $\lambda$ for fixed $\beta = 0.5$ (the dashed line shows the actual $\lambda$ value for the Web). On the right, the same as a function of $\beta$ for $\lambda = 1.36$ (the dashed lines enclose the typical $\beta$ values). In both cases we use the standard $z = 1$.

To show how retrieval times are affected by a non-uniform distribution when we have to traverse the matching blocks, we do the analysis for the document size distribution of the Web. As we have shown, if a block has size $x$ then the probability that it has to be traversed is $(1 - e^{-x/n^{\beta-\nu}})$. We multiply this by the cost $x$ to traverse it and integrate over all the possible sizes, so as to obtain its expected traversal cost (recall Eq. (8.1))

$$\int_k^\infty x(1 - e^{-x/n^{\beta-\nu}})p(x)dx$$

which we cannot solve[3]. However, we can separate the integral in two parts. $(a)$ $x = o(n^{\beta-\nu})$ and $(b)$ $x = \Omega(n^{\beta-\nu})$. In the first case the traversal probability is $O(x/n^{\beta-\nu})$ and in the second case it is $\Omega(1)$. Splitting the integral in two parts we obtain

$$\Theta\left(\frac{\lambda k^\lambda}{2-\lambda}\, n^{(\beta-\nu)(1-\lambda)} + \frac{\lambda k^\lambda}{\lambda - 1}\, n^{(\beta-\nu)(1-\lambda)}\right) \;=\; \Theta\left(\frac{\lambda k^\lambda}{(2-\lambda)(\lambda-1)}\, n^{(\beta-\nu)(1-\lambda)}\right)$$

Now that we have the cost per block, we multiply by $r = (\lambda - 1)/(\lambda k)\ n$ (Eq. (8.7)) to obtain the total amount of work. This is

$$\frac{k^{\lambda-1}}{2-\lambda}\, n^{1-(\lambda-1)(\beta-\nu)}$$

On the other hand, if we used blocks of fixed size, the time complexity (using Eq. (8.2)) would be $O(bn^{1-\beta+\nu})$, where $b = zb^*$. The ratio between both search times is

$$\frac{\text{doc. index traversal}}{\text{block index traversal}} = \frac{(\lambda-1)n^{(2-\lambda)(\beta-\nu)}}{\lambda(2-\lambda)zk^{2-\lambda}}$$

which shows that the document index would be asymptotically slower than a block index as the text collection grows. In practice, the ratio is between $O(n^{0.2})$ and $O(n^{0.4})$. The value of $z$ is not important here since it is a constant.

### 8.3.3 Experimental Validation

In this section we validate experimentally the previous analysis. The collection is considered as a single large file, which is logically split into blocks of fixed size. The larger the blocks, the faster to build and the smaller the index, but also the larger the proportion of text to search sequentially at query time. We measure $n$ and $b$ in bytes, not in words.

#### 8.3.3.1 Fixed Block Size

We show the space overhead of the index and the time to answer queries for three different fixed block sizes: 2 Kb, 32 Kb and 512 Kb. See Figure 8.10. Observe that the time is measured in a machine-independent way, since we show the percentage of the whole text that is sequentially searched. Since the processing time in the vocabulary is negligible, the time complexity is basically proportional to this percentage. The decreasing percentages indicate that the time is sublinear.

The queries are the same used to measure the amount of matching in the vocabulary, again ensuring at most 5% of error with a confidence level of 95%. Using least squares we obtain that the amount of traversed text is $0.10n^{0.79}$ for $b = 2$ Kb, $0.45n^{0.85}$ for $b = 32$ Kb, and $0.85n^{0.89}$ for $b = 512$ Kb. In all cases, the relative error of the approximation is under 5%. As expected from the analysis, the space overhead becomes linear (since $\gamma = 1$) and the time is sublinear (the analysis predicts $O(n^{0.78})$, which is close to these results).

We observe that the analysis is closer to the curve for smaller $b$. This is because the fact that $b = O(1)$ shows up earlier (i.e. for smaller $n$) when $b$ is smaller. The curves with larger $b$ should converge to the same exponents for larger $n$.

[3]Which is more convincing, Maple cannot solve.

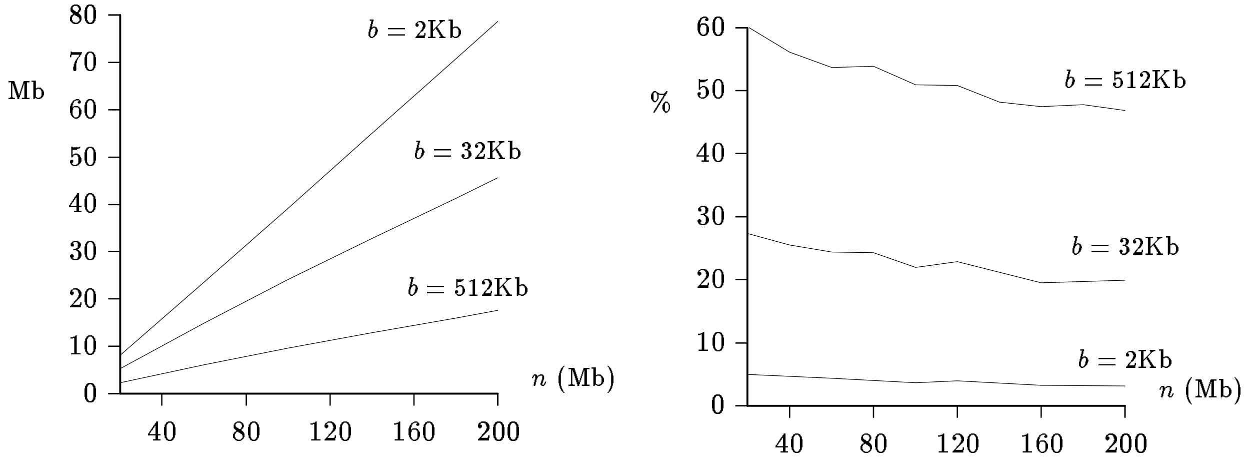


Figure 8.10: Experiments for fixed block size $b$. On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched.

#### 8.3.3.2 Fixed Number of Blocks

To show the other extreme, we take the case of fixed $r$. The analysis predicts that the time should be linear and the space should be sublinear (more specifically, $O(n^{1/\theta}) = O(n^{0.53})$). This is the model used in *Glimpse* for the tiny index (where $r \approx 256$).

See Figure 8.11, where we measure again space overhead and query times, for $r = 2^8$, $2^{12}$ and $2^{16}$. Using least squares we find that the space overhead is sublinear in the text size $n$. For $r = 2^8$ we have that the space is $0.87n^{0.53}$, for $r = 2^{12}$ it is $0.78n^{0.75}$, and for $r = 2^{16}$ it is $0.74n^{0.87}$. The relative error of the approximation is under 3%. As before, the analysis is closer to the curve for smaller $r$, by similar reasons (the effect is noticed sooner for smaller $r$).

On the other hand, the percentage of the traversed text increases. This is because the proportion of text traversed (Eq. (8.1)) is $(1 - e^{-n^{1-\beta+\nu}})$, which tends to 1 as $n$ grows.

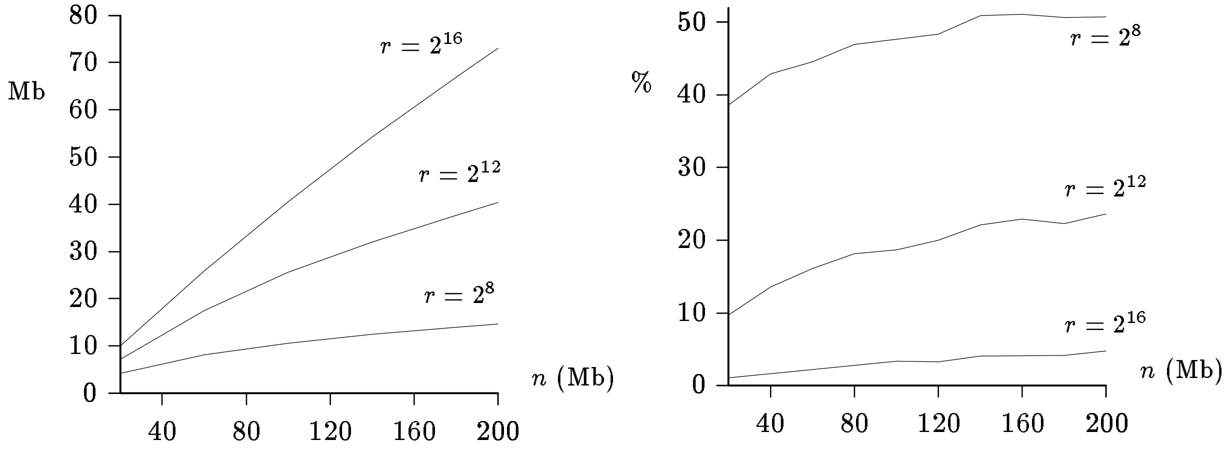


Figure 8.11: Experiments for fixed number of blocks $r$. On the left, space taken by the indices. On the right, the right, percentages of the text sizes sequentially searched.

#### 8.3.3.3 Sublinear Space and Time

Finally, we show experimentally in Figure 8.12 that time and space can be simultaneously sublinear. We test $\gamma = 0.92$, 0.94 and 0.96. The analysis predicts the values shown in the table of Figure 8.8. Using least squares we find that the space overhead is sublinear and very close to the predictions: $0.40n^{0.89}$, $0.41n^{0.92}$ and $0.42n^{0.95}$. The error of the approximations is under 1%.

The percentage of the traversed text decreases, showing that the time is also sublinear. The least squares approximation shows that the query times for the above $\gamma$ values are $0.24n^{0.95}$, $0.17n^{0.94}$ and $0.11n^{0.91}$, respectively. The relative error is smaller than 2%.

Hence, we can have for this text an $O(n^{0.94})$ space and time index (our analysis predicts $O(n^{0.93})$).

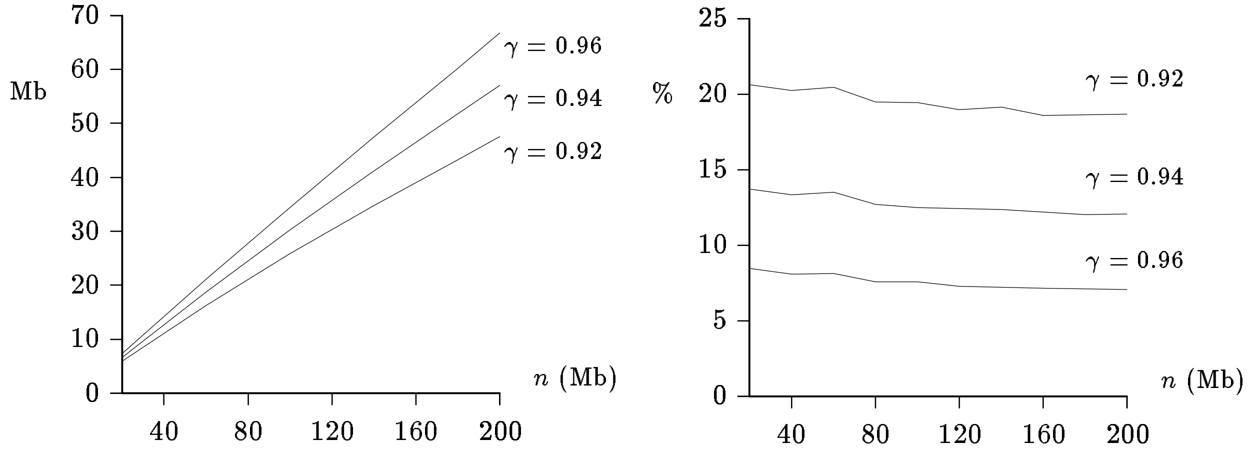


Figure 8.12: Experiments for fixed $\gamma$ (simultaneous sublinearity). On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched.

As another example, we give in Figure 8.13 the results on simultaneous sublinearity for the ZIFF collection, which has near 220 Mb after filtering. The values for this collection are $\beta = 0.51$ and $\theta = 1.79$. Least squares show a very good agreement with the analysis: we have $0.71n^{0.92}$ for $\gamma = 0.92$, $0.60n^{0.94}$ for $\gamma = 0.94$ and $0.55n^{0.95}$ for $\gamma = 0.96$. The relative error is below 0.5%. The times give $0.22n^{0.99}$ for $\gamma = 0.92$, $0.17n^{0.98}$ for $\gamma = 0.94$ and $0.14n^{0.96}$ for $\gamma = 0.96$. Hence, we can have an $O(n^{0.96})$ space and time index for ZIFF. It is interesting to notice that, although ZIFF has a larger vocabulary than WSJ, the results are not better. This is because the number of matching words in the vocabulary is also higher.

## 8.4 Improving the Search Algorithms

In this last section we study different improvements to the search algorithm typically used for approximate searching on inverted indices. First we show how the vocabulary search can be improved (only for approximate searching) and later how the search on the blocks, if necessary, can also be improved (this works for regular expressions and extended patterns as well).

Unfortunately the two optimizations hardly can be applied in conjunction. When the indexing scheme makes it necessary to traverse the blocks, the gains due to faster vocabulary search are totally marginal (less than 1%). Faster vocabulary search is of interest when either full inversion is used or block traversal is not necessary, and therefore the vocabulary search dominates the overall search time.

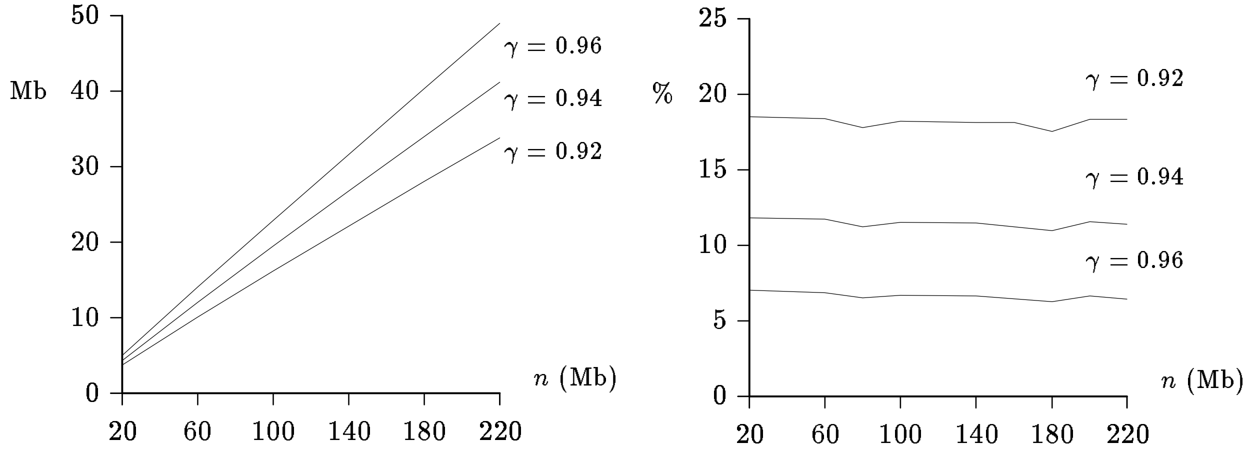


Figure 8.13: Experiments for fixed $\gamma$ (simultaneous sublinearity) for the ZIFF collection. On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched.

### 8.4.1 Vocabulary Search

Until now, we have relied on an on-line search in the vocabulary of the text. To reduce the time of that on-line search, we show now how to index the vocabulary as a metric space. We show that with reasonable space overhead we can improve by a factor of two over the fastest on-line algorithms, when the tolerated error level is low (which is reasonable in text searching).

Since the vocabulary of the text is quite small compared to the text itself, an on-line search on it takes a few seconds at most. While this may be adequate for the case of a single end-user, it is interesting to improve the search time for other scenarios. For example, the algorithm could be a module of another more complex package (say, a linguistic analyzer) and receive thousands of search requests. As another example, consider a a Web search engine which receives many requests per second and cannot spend even a few seconds to traverse the vocabulary. In this section we organize the vocabulary as a metric space using the edit distance function *ed*(), and use a known data structure to index such spaces. This idea may also have other applications where a dictionary of words is searched allowing errors, such as in spelling problems.

The technique that we present needs to compute the exact edit distance among strings, and therefore it relies on the classical algorithm. The result is that, although it may perform a few evaluations of the edit distance (say, 5% of the whole vocabulary), it may be slower than an on-line traversal with a fast algorithm. On the other hand, many of the fastest algorithm could not be usable if some extension over the edit distance was desired, while the classical algorithm (and hence our technique) can accommodate many extensions at no extra cost.

#### 8.4.1.1 Searching in General Metric Spaces

The concept of "approximate" searching has applications in a vast number of fields. Some examples are images, fingerprints or audio databases; machine learning; image quantization and compression; text retrieval (for approximate string matching or for document similarity); genetic databases; etc.

All those applications have some common characteristics. There is a universe $U$ of *objects*, and a nonnegative *distance function* $d : U \times U \longrightarrow R^+$ defined among them. This distance satisfies the three axioms that makes the set a *metric space* (see Section 2.1). The smaller the distance between two objects, the more "similar" they are. This distance is considered expensive to compute (e.g. comparing two fingerprints). We have a finite *database* $S \subseteq U$, which is a subset of the universe of objects and can be preprocessed (to build an index, for instance). Later, given a new object from the universe (a *query* $q$), we must retrieve all similar elements found in the database. There are different queries depending on the application, but the simplest one is: given a new element $q$ and a maximum distance $k$, retrieve all the elements in the set which are at distance at most $k$ from $q$.

This is applicable to our problem because we have a set of elements (the vocabulary) and the distance $ed()$ satisfies the stated axioms. A number of data structures exist to index the vocabulary so that the queries can be answered without inspecting all the elements. Our distance is *discrete* (i.e. gives integer answers), which determines the data structures which can be used. We briefly survey the main applicable structures now.

The first proposed structure is the Burkhard-Keller Tree (or BK-tree) [BK73], which is defined as follows: an arbitrary element $a \in S$ is selected as the root, whose subtrees are identified by integer values. In the $i$-th children we recursively build the tree for all elements in $S$ which are at distance $i$ from $a$. This process can be repeated until there is only one element to process, or there are no more than $b$ elements (and we store a *bucket* of size $b$).

To answer queries of the form $(q, k)$, we begin at the root and enter into all children $i$ such that $d(a,q) - k \leq i \leq d(a,q) + k$, and proceed recursively (the other branches are discarded using the triangular inequality). If we arrive to a leaf (bucket of size one or more) we compare sequentially all the elements. We report all the elements $x$ found that satisfy $d(q,x) \leq k$.

Another structure is called "Fixed-Queries Tree" or FQ-tree [BYCMW94]. This tree is basically a BK-tree where all the elements stored in the nodes of the same level are the same (and of course do not necessarily belong to the set stored in the subtree), and the real elements are all in the leaves. The advantage of such construction is that some comparisons are saved between the query and the nodes along the backtracking that occurs in the tree. If we visit many nodes of the same level, we do not need to perform more than one comparison per level. This is at the expense of somewhat taller trees. Another variant is proposed in [BYCMW94], called "Fixed-Height FQ-trees", where all the leaves are at the same depth $h$, regardless of the bucket size. This makes some leaves deeper than necessary, which makes sense because we may have already performed the comparison between the query and one intermediate node, therefore eliminating for free the need to compare the leaf. In [Sha77], an intermediate structure between BK-trees and FQ-trees is proposed.

An analysis of the performance of FQ-trees is presented in [BYCMW94], which disregarding some complications can be applied to BK-trees as well. They basically show that the number of traversed nodes is $O(n^x)$, where $n$ is the size of the set and $0 < x < 1$ depends on the metric space. It can also be shown that the fixed-height variant traverses a sublinear number of nodes [BYN98b].

Some approaches designed for continuous distance functions , e.g. [Uhl91, Yia93, Bri95, FL95], are not covered in this brief review. The reason is that these structures do not use all the information obtained from the comparisons, since this cannot be done in continuous spaces. This is, however, done in discrete spaces and this fact makes the reviewed structures superior to those for continuous spaces, although they would not be directly applicable to the continuous case. We also do not cover algorithms which need $O(n^2)$ space such as [Vid86] because they are impractical for our application.

It is not the aim of this thesis to improve on this techniques for general metric spaces. We just make use of the available ones for our goals.

### 8.4.1.2 The Vocabulary as a Metric Space

Traversing the whole vocabulary on-line is like comparing the query against the whole database in a metric space. Our proposal is to organize the vocabulary such as to avoid the complete on-line traversal. This organization is based on the fact that we want, from a set of words, those which are at edit distance at most $k$ from a given query. The edit distance $ed()$ used satisfies the axioms which make it a metric, in particular a discrete metric.

The proposal is therefore, instead of storing the vocabulary as a sequence of words, organize it as a metric space using one of the available techniques. The distance function to use is $ed()$, which is computed by dynamic programming in time $O(m_1 m_2)$, where $m_1$ and $m_2$ are the lengths of the two words to compare. Although this comparison takes more than many efficient algorithms, it will be carried out only a few times to get the answer. On the other hand, the dynamic programming algorithm is very flexible to add new editing operations or changing their cost, while the most efficient on-line algorithms are not that flexible.

Figure 8.14 shows our proposed organization. The vocabulary is stored as a contiguous text (with separators among words) where the words are sorted. This allows exact or prefix retrieval by binary search, or another structure can be built onto it. The search structure to allow errors goes on top of that array and allows approximate or exact retrieval.

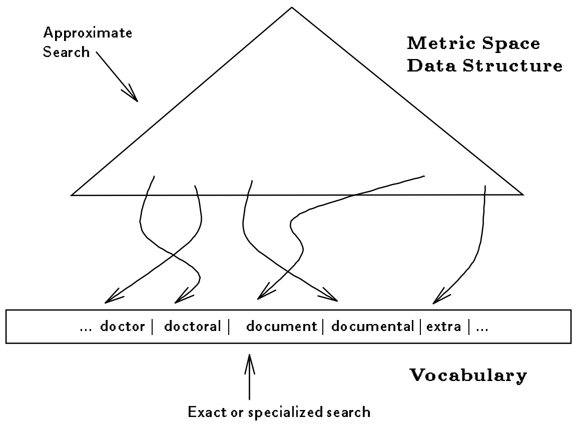

Approximate Search
Metric Space Data Structure
... doctor | doctoral | document | documental | extra | ...
Vocabulary
Exact or specialized search

Figure 8.14: Proposed data structure.

An important difference between the general assumptions and our case is that the distance function is not so costly to compute as to make negligible all other costs. For instance, the space overhead and non-locality of accesses incurred by the new search structures could eliminate the advantage of comparing the query against less words in the vocabulary. Hence, we do not consider simply the number of comparisons but the complete CPU times of the algorithms, and compare them against the CPU times of the best sequential search algorithms run over the complete vocabulary. Moreover, the efficiency in all cases depends on the number of errors allowed (all the algorithms worsen if more errors are allowed). We have also to consider the extra space incurred because the vocabulary is already large to fit in main memory. Finally, although the asymptotic analysis of the Appendix shows that the number of traversed nodes is sublinear, we must verify how does this behave for the vocabulary sizes which are used in practice.

It is interesting to notice that any structure to search in a metric space can be used for exact searching, since we just search allowing zero errors (i.e. distance zero). Although not as efficient as data structures designed specifically for exact retrieval (such as hashing or binary search), the search times may be so low that the reduced efficiency is not as important as the fact that we do not need an additional structure for exact search (such as a hash table).

#### 8.4.1.3 Experimental Results

We tested two different structures: BK-trees (BKT) and FQ-trees (FQT). We tested buckets of size 1, 10 and 20. As explained before, other structures for metric spaces are not well suited to this case (we verified experimentally this fact). We used the 500,000 words (5 Mb) vocabulary of the English TREC collection (1 Gb). The vocabulary was randomly permuted and separated in 10 incremental subsets of size 50,000 to 500,000.

Our first experiment deals with space and time overhead of the data structures that implement the search in a metric space, and its suitability for exact searching. Figure 8.15 shows the results. As it can be seen, build times are slightly superlinear ($O(n \log n)$ in fact, since the height is $O(\log n)$). The overhead to build them is normally below 2 minutes, which is a small percentage (10% at most) of the time normally taken to build an index for a 1 Gb text database.

If we consider extra space, we see that the BKT poses a fixed space overhead, which reaches a maximum of 115% for $b = 1$. This corresponds to the fact that the BKT stores at most one node per element. The space of the FQT is slightly superlinear (the internal nodes are empty) and for this experiment is well above 200% for $b = 1$.

Finally, we show that the work to do for exact searching involves a few distance evaluations (20 or less) with very low growth rate (logarithmic). This shows that the structure can be also used for exact searching.

We show in Figure 8.16 the query performance of the indices to search with one error. As it can be seen, no more than 5-8% of the dictionary is traversed (the percentage is decreasing since the number of comparisons are sublinear). The user times correspond quite well to the number of comparisons. We show the percentage of user times using the structures versus the best on-line algorithm for this case (the one presented in Section 6.1). As it can be seen, for the maximum dictionary size we reach 40% of the on-line time for the best metric structures. From those structures, we believe that BKT with $b = 1$ is the best choice, since it is faster than all the FQT's (and takes less space). Another alternative which takes less space (close to 70%) is BKT with $b = 10$, while it achieves 60% of the times of on-line searching.

The result for two errors (not shown) is not so good. This time the metric space algorithms do not improve the on-line search, despite that the best ones traverse only 17%-25% of the vocabulary. The reason is that the off-line algorithms are much more sensitive to the error level than the on-line ones. This shows that our scheme is only useful to search with one error.

Table 8.2 shows the results of the least squares fitting over the number of comparisons performed by the different data structures. For $k = 0$ we obtain a good logarithmic approximation, while the bucket size seems to affect the constant rather than the multiplying factor.

For $k = 1$, the results confirm the fact that the structures inspect a sublinear number of nodes. Notice that the exponent is smaller for BKT than for FQT, although the last ones have a better constant. The constant, on the other hand, seems to keep unchanged when the bucket size varies (only the exponent is affected). This allows extrapolating that BKT will continue to improve over FQT for larger data sets (it is well known that all the conclusions about metric space data structures

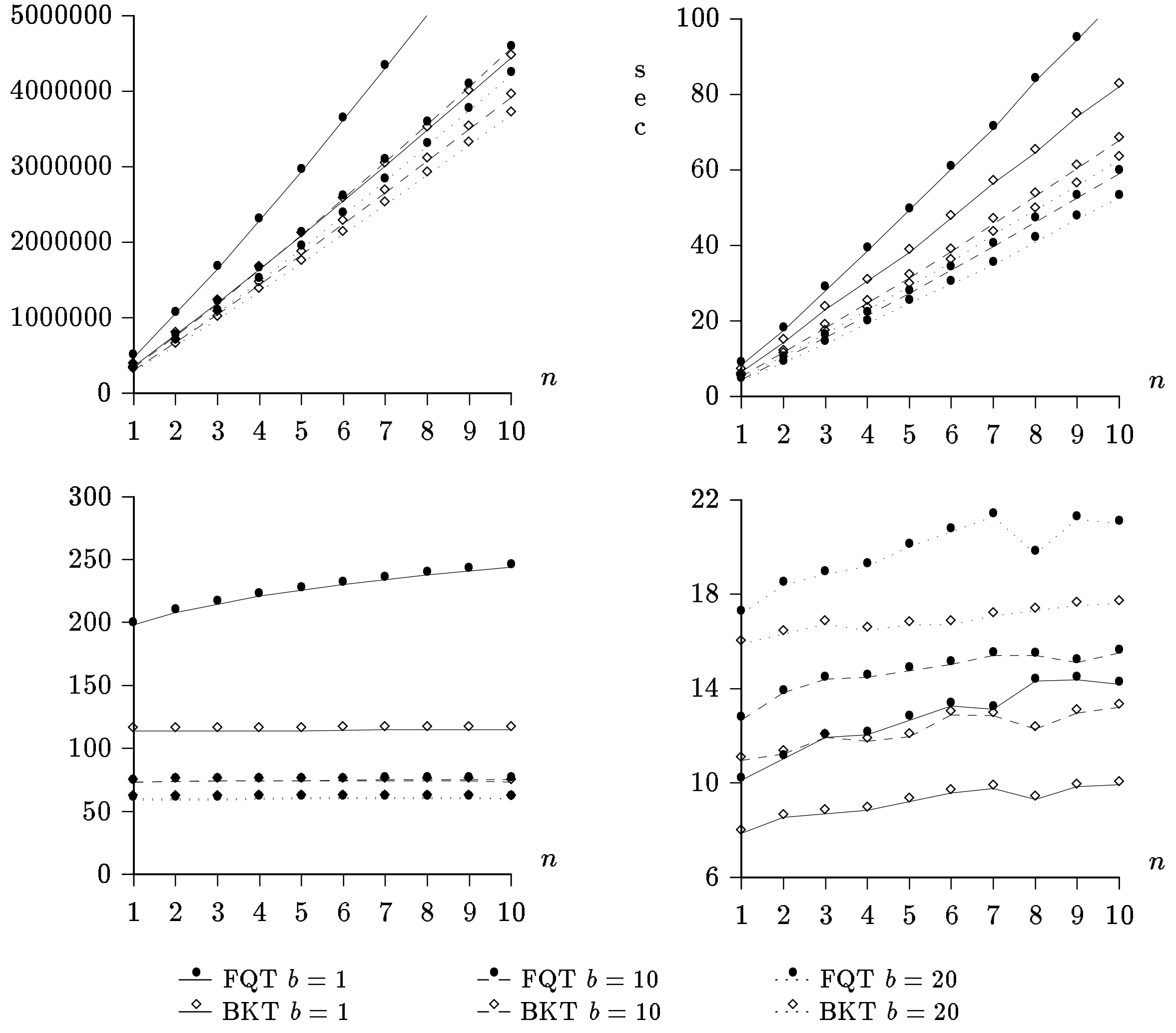


Figure 8.15: Comparison of the data structures. From top to bottom and left to right, number of distance evaluations and user times to build them, extra space taken over the vocabulary size, and number of distance evaluations for exact search. The $x$ axis is expressed in multiples of 50,000.

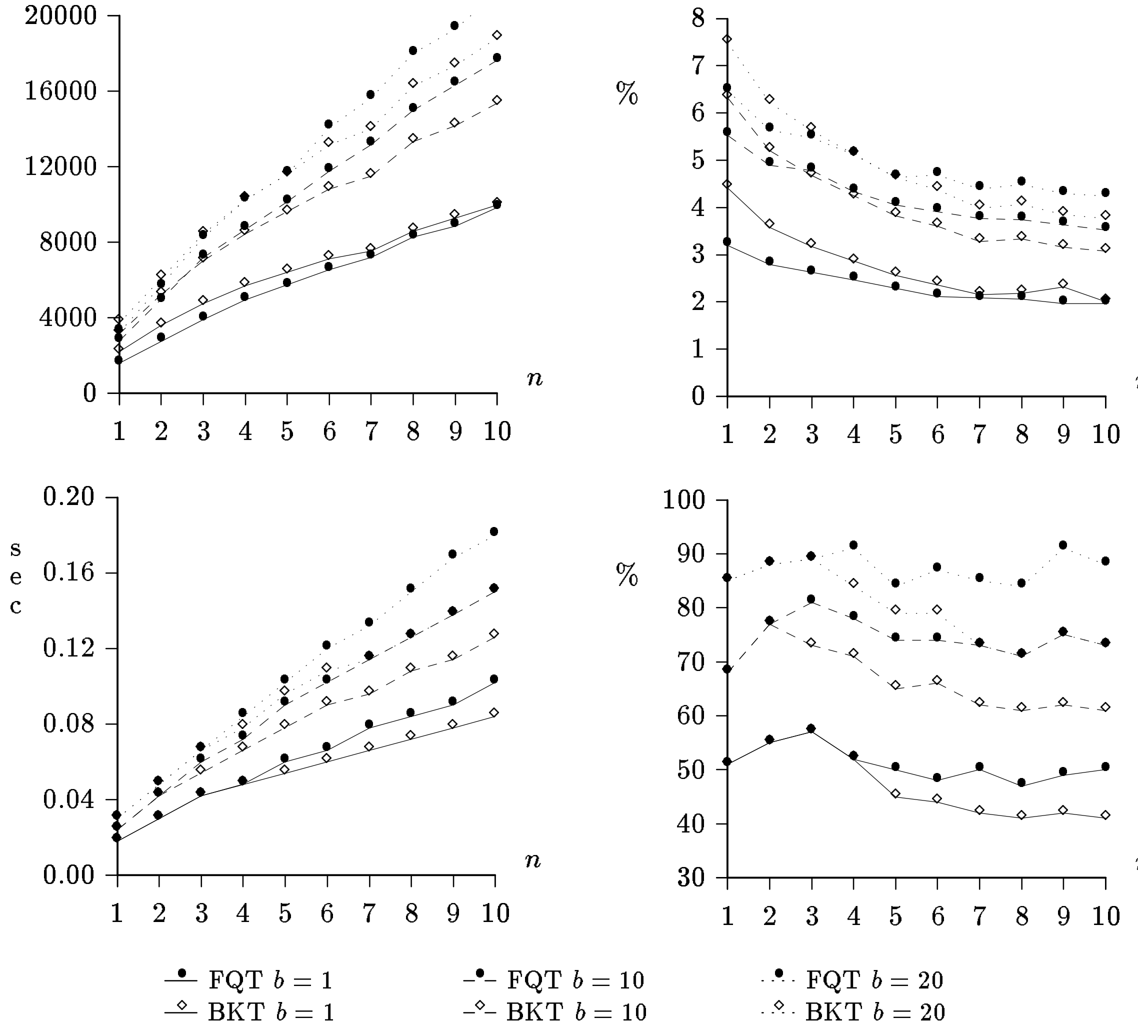


Figure 8.16: Search allowing one error. The first row shows the number of comparisons (on the left, absolute number, on the right, percentage over the whole dictionary). The second row shows user times (on the left, seconds, on the right, percentage over the best on-line algorithm). The $x$ axis is expressed in multiples of 50,000.

| Structure | $k=0$ | $k=1$ | $k=2$ |
|---|---|---|---|
| BKT ($b=1$) | $0.87\ln(n) - 1.52$ | $2.25\ n^{0.639}$ | $1.91\ n^{0.822}$ |
| BKT ($b=10$) | $0.96\ln(n) + 0.39$ | $2.21\ n^{0.673}$ | $1.52\ n^{0.859}$ |
| BKT ($b=20$) | $0.69\ln(n) + 8.36$ | $2.16\ n^{0.691}$ | $1.42\ n^{0.871}$ |
| FQT ($b=1$) | $1.91\ln(n) - 10.84$ | $0.36\ n^{0.777}$ | $0.54\ n^{0.926}$ |
| FQT ($b=10$) | $1.17\ln(n) + 0.26$ | $0.50\ n^{0.798}$ | $0.63\ n^{0.921}$ |
| FQT ($b=20$) | $1.73\ln(n) - 1.58$ | $0.49\ n^{0.814}$ | $0.69\ n^{0.919}$ |

Table 8.2: Least squares fitting for the number of comparisons made by the different data structures.

depend strongly on the particular space and distance function, so this does not allow a generalization to other cases). The results for $k = 2$ increase the exponent (which will be close to 1 for $k = 3$). The relative error is between 15% and 20% in all cases.

The least squares fitting over the real CPU times give similar growth rates, for instance it is $O(n^{0.65})$ for BKT ($b = 1$).

Our implementation of the BK-trees is not optimized for space. We estimate that with a careful implementation the overhead can be reduced from 100% to 65%. This overhead is quite reasonable in most cases.

The only alternative structure we are aware of is [SM96], where the vocabulary is organized into a trie data structure. In the trie, each node represents a unique substring of the vocabulary. The dynamic programming algorithm is run on the trie instead of on the plain vocabulary, therefore eliminating the possible substring repetitions that appear (a very similar technique is proposed in [Ukk93, Cob95] for sequence-oriented indices, see Section 3.3.2). The search on the trie is abandoned as soon as it can be seen that a match with $k$ errors is not possible.

We have implemented this scheme, obtaining very fast construction times (7 seconds for the 5 Mb vocabulary) and very fast search times (for the maximum size, 0.0054 seconds for $k = 1$ and 0.0669 for $k = 2$), which outperforms our scheme by an order of magnitude. However, the main problem of tries is their high space overhead. Naive implementations may pose 400% or more overhead over the text size.

We have carefully implemented a version that optimizes space, and obtained 100% extra overhead over the text. This is similar to the overhead we have presented for the BK-trees. However, our implementation of the BK-trees is not optimized for space. With a careful implementation can obtain only 65% overhead.

We are currently working on new data structures to index metric spaces which could dramatically improve the effectiveness of the current scheme. However, these are not yet finished and are not part of this thesis.

### 8.4.2 Block Search

We propose a new strategy for approximate searching on block addressing indices, which we experimentally find 4-5 times faster than *Glimpse*, and that unlike *Glimpse*, takes advantage of the vocabulary information even when the whole text has to be verified.

We also start the search by sequentially scanning the vocabulary with an on-line approximate search algorithm. Once the blocks to search have been obtained, *Glimpse* uses *Agrep* (i.e. approximate searching) again over the blocks. However, this can be done better.

Since we have run an on-line search over the vocabulary first, we know not only which blocks contain an approximate match of the search pattern, but also *which words* of the vocabulary matched the pattern and are present in each block. Hence, instead of using again approximate search over each block as *Glimpse*, we can run an *exact* (multipattern) search for those matching words found in the vocabulary. Currently we perform a single search with all the patterns involved over all the matching blocks. As future work we plan to perform a different search on each block, so that only the patterns present in that block are searched. In most cases, this can be done much more efficiently than approximate searching. Moreover, since as we show later, most of the search time is spent in the search of the text blocks (if this is needed), this improvement has a strong impact on the overall search time.

We use an extension of the Boyer-Moore-Horspool-Sunday algorithm [Sun90] to multipattern search. This gave us better results than an Aho-Corasick machine, since as shown in Figure 8.1, few words are searched on each block (this decision is also supported by [Wat96]).

We compared this strategy against *Glimpse* version 4.0. We used the "small" index provided by *Glimpse*, i.e. the one addressing files (i.e. the sequential search must be done on the matching files). Our index used also files as the addressing unit for this comparison. The tests were run on a Sun SparcServer 1000 with 128 Mb of RAM, running Solaris 2.5, which was not performing other tasks. However, only 4 Mb of RAM were used by the indexers. We used the WSJ collection.

The stopword mechanism and treatment of upper and lower-case letters is somewhat particular in *Glimpse*. We circumvent this problem by performing all filtering and stopword elimination directly in the source files, and then using both indices without filtering or stopwords considerations.

Our indexer took near 16 minutes to index the collection (i.e. more than 10 Mb per minute), while *Glimpse* took 28 minutes. This is due to different internal details which are not of interest to this work, e.g. the indexers have different capabilities apart from approximate searching. Both indices took approximately 7 Mb. This is less than 3% of the size of the collection (this low percentage is because the files are quite large).

We are not comparing the complete indexing mechanisms, but only their strategy to cope with approximate search of words when they have to be sequentially searched on the text. Issues such as a different addressing granularity will not change the proportion between the search times.

In both indices we retrieve whole words that match the pattern. This is the default in this chapter and we believe that this option is more natural to the final user than allowing subword matching (i.e. "sense" matching with *one* error in "consensus").

Table 8.3 (upper part) shows the times obtained (user times). As it can be seen, the mechanism we propose is 4-5 times faster in practice (i.e. taking into account all the processing needed). We also show the percentage of the text sequentially inspected and the average number of matches found, as well as the number of words matching in the vocabulary. We can see that an important part of the text is inspected, even for queries with acceptable precision (this is because the files are large). Moreover, the times are almost proportional to the amount of sequential search done (we process near 5 Mb/sec, while *Glimpse* is close to 1 Mb/sec). Therefore, the advantage of searching with a multipattern exact search instead of an approximate search algorithm is evident. Even if the whole text is searched (in which case *Glimpse* is not better than *Agrep*, i.e. a complete sequential search), our indexing scheme takes advantage of the vocabulary, because it never searches the text for an approximate pattern.

| Errors | Ours | *Glimpse* | Ours/*Glimpse* | % inspected | # matches | # vocab. matches |
|---|---|---|---|---|---|---|
| Matching Complete Words | | | | | | |
| 1 | 8.20 | 34.99 | 23.42% | 24.94% | 871.86 | 4.97 |
| 2 | 18.05 | 82.50 | 21.91% | 43.83% | 2591.02 | 25.54 |
| 3 | 29.37 | 143.69 | 20.43% | 77.81% | 7341.84 | 31.15 |
| Subword Matching Allowed | | | | | | |
| 1 | 39.37 | 16.05 | 245.30% | 41.45% | 44541.50 | 159.07 |
| 2 | 73.04 | 64.12 | 113.91% | 64.28% | 44991.80 | 230.48 |
| 3 | 75.84 | 132.56 | 57.21% | 77.39% | 31150.50 | 182.92 |

Table 8.3: Times (in seconds) and other statistics to retrieve all occurrences of a random word with different number of errors.

Table 8.3 (lower part) presents the case in which subword matching is also allowed. The precision is much lower (i.e. there are more matches), which shows that this query is unlikely to be interesting for the users. It can also be seen that much more text is traversed. The performance of our algorithm degrades due to a larger amount of words matching in the vocabulary, which reduces the effectiveness of our multipattern exact searching against plain approximate search (this should improve when we implement a separate search per block with only the patterns that matched that block). On the other hand, *Glimpse* improves for one and two errors because of specialized internal algorithms to deal with this case. The net result is that our algorithm is slower for one and two errors, although it is still faster for three errors. This test shows that our approach is better when not too many words match in the vocabulary, which is normally the case of useful queries.

The idea of multipattern search on the blocks is in fact the essence of the success of *Cgrep* [MNZBY98b, MNZBY98a]. *Cgrep* is not an inverted file but a compressor able to efficiently search from simple patterns to regular expressions, allowing or not errors. It is based on Huffman coding on words. Since the words are the symbols of the coder, a table with all the words (that is, a vocabulary!) is stored together with the compressed file. The search starts in the vocabulary, much as in our inverted indices. Once the matching words are obtained, their compressed codes are searched in the compressed text. Although there is no indexing and all the text has to be traversed, the search is a multipattern exact search for the compressed codes of the matching words, which can be much faster than the original search (e.g. an approximate search). This allows *Cgrep* to be faster than *Agrep*. As future work we plan to integrate our block addressing index with the compression scheme of *Cgrep*, so that the index and the text are compressed and the search on the compressed blocks is done using *Cgrep*.

# Chapter 9

# Sequence-Retrieving Indices

In this chapter we consider indices able to solve the general approximate string matching problem, i.e. to retrieve any text sequence which is at distance $k$ or less from the pattern. As shown in Chapter 2, there are two main types of sequence-retrieving indices. We present new ideas in both areas, as well as novel analytical results. The results of this chapter have been published in [BYN97c, NBY98c, BYNST97].

## 9.1 An Index Based on Sampling

We propose a sequence-retrieving indexing technique which is practical and powerful, and especially optimized for natural language text. It is an inverted index built on the vocabulary of all the text substrings of length $q$, and the search is an off-line version of Section 6.1 (i.e. the pattern is split in $k+1$ pieces which are searched with no errors). We design an algorithm to optimize the pattern partition so that the total number of verifications to perform is minimized (this idea does not work so well in on-line searching, as explained in Section 6.1). This is especially useful for natural language texts and allows knowing in advance the expected cost of the search and the expected relevance of the query to the user. We show experimentally the performance of the index, finding that it is a practical alternative for text retrieval: the space overhead is between two and four times the text size, and for useful queries the retrieval times are reduced from 10% to 60% of the best on-line algorithm.

### 9.1.1 Indexing Text Substrings

At indexing time, we select a fixed length $q$. Every text $q$-gram (substring of length $q$) is stored in the index (in lexical order). To resemble traditional inverted lists, we call *vocabulary* the set of all different $q$-grams. The number of different $q$-grams is denoted $V$, which is $\leq n$ (in a text of $n$ characters there are $n-q+1$ $q$-grams, but only $V$ different $q$-grams) [1]. Together with each $q$-gram, we store the list of the text positions where it appears, in ascending positional order as in a traditional inverted index (see Section 2.10). Figure 9.1 shows a small example.

If block addressing (Section 2.10) is used, the text is divided in blocks of a fixed length $b$, and all the $q$-grams that start in the block are considered to lie inside the block. Only the ascending list

[1]For the correctness of the algorithms, it is necessary that the last $q-1$ suffixes of the text are entered as $q$-grams too, even when they are of length $< q$.

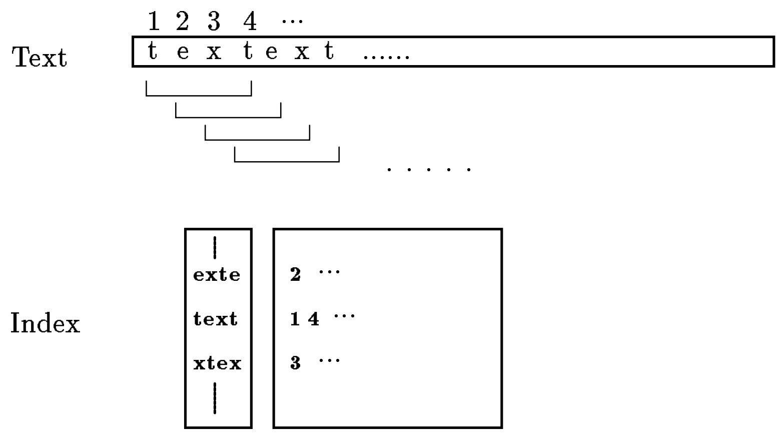


Figure 9.1: The indexing scheme for $q = 4$.

of the blocks where each $q$-gram appears is stored in this case. This makes the index smaller (since there is only one reference for all the occurrences of a $q$-gram in a single block, and also the pointers to blocks can be smaller).

To search a pattern of length $m$ with $k$ errors, we split the pattern in $k+1$ pieces, search each piece in the index of $q$-grams of the text, and merge all the occurrences of all the pieces, since each one is a candidate position for a match. The neighborhood of each candidate position is then verified with a sequential algorithm as in Section 6.1. If blocks are used, each candidate block must be completely traversed with an on-line algorithm. Figure 9.2 illustrates the search process.

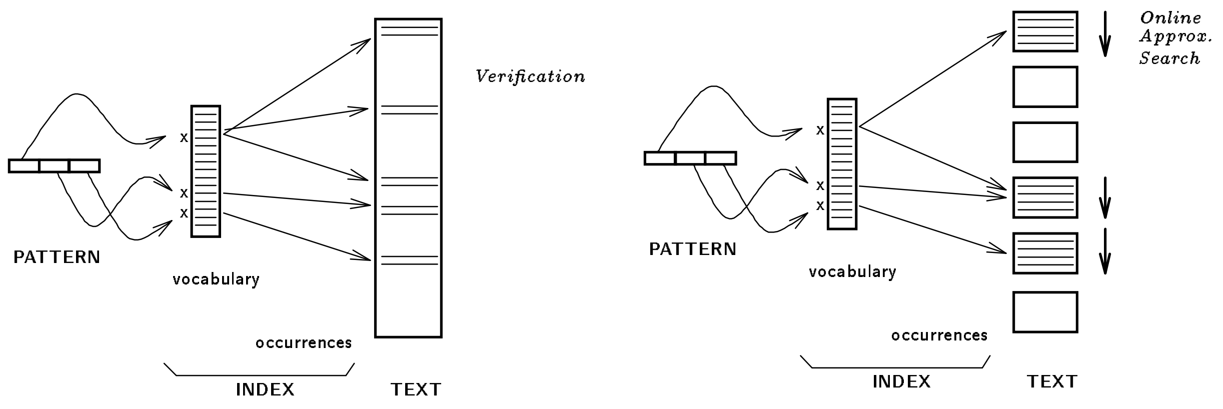


Figure 9.2: The search process, with exact addressing and block addressing.

Of course the pattern pieces may not have the same length $q$. If a piece is shorter than $q$, all the $q$-grams with the piece as prefix are to be considered as occurrences of the piece (they are contiguous in the index of $q$-grams). If the piece is longer, it is simply truncated to its first $q$ characters (it is possible to verify later, in the text, whether the $q$-gram starts in fact an occurrence of the piece before verifying the whole area).

When the pattern is split in $k+1$ pieces, we are free to select those pieces as we like. We explain this idea in the on-line algorithm (Section 6.1, see also Section 4.2): knowing or assuming a given character distribution for the text to search, the pieces are selected so that the probabilities of all pieces are similar. This minimizes the total number of verifications to perform, on average.

Although in Section 6.1 we use this technique, there are two drawbacks in that case: first, we have to estimate the probabilities of the pieces (in that case we multiply the individual character probability); and second, the Boyer-Moore-like search algorithm that we use degrades as the shortest piece gets shorter. Hence, in that case we include the search cost in the cost model and the results are positive but modest.

We can do much better here. The key point is that it is very cheap to compute in advance the *exact* number of verifications to perform for a given piece. We just locate the piece in the q-gram index with binary search. In the general case we obtain a contiguous region, for pieces shorter than $q$. By storing, for each $q$-gram, the *accumulated* length of the lists of occurrences, we can subtract the lengths at the endpoints of the region to obtain immediately the number of verifications to perform. The complete process takes $O(\log V) = O(\log n)$.

We adapt the dynamic programming algorithm of Section 6.1 that computes the optimal partition. As a side result, we know in advance the total cost to pay to retrieve the results, which as explained is useful as early feedback to the user. The only difference is in the computation of $R[i,j]$, which is carried out as explained in the previous paragraph. This takes $O(m^2 \log n)$ if the pieces are binary searched. As explained in the original formulation of the algorithm, we need $O(m^2)$ space and $O(m^2 k)$ time to build the other tables. If we replace the binary search by a trie of $q$-grams, the time to build the initial $R$ table can be lowered to $O(m^2)$: for each $i$ we start at the root of the trie and find the $q$-grams corresponding to $R[i,i]$, $R[i,i+1]$, $R[i,i+2]$, and so on. Each new cell is found in $O(1)$ time, and there are $O(m^2)$ cells. Hence the total cost of this optimization can be as low as $O(m^2 k)$.

Notice that, since it is possible that $q$ is small, we may select pieces of the pattern whose beginnings are farther apart than $q$ characters, and therefore the $k+1$ pieces are non-contiguous. This makes use of the $x_r$ of the Partitioning Lemma of Section 4.2.

### 9.1.2 Analysis

We analyze the time and space requirements of our index, as well as its retrieval performance.

#### 9.1.2.1 Building the Index

To build the index we scan the text in a single pass, using hashing to store all the $q$-grams that appear in the text. This $q$ must be selected as large as possible, but small enough for the total number of such $q$-grams to be small (practical values for natural language text are $q = 3..5$).

Although we scan every $q$-gram and any good hash function of a $q$-gram takes $O(q)$ time, the total expected time is kept $O(n)$ instead of $O(nq)$ by using a technique similar to Karp-Rabin [KR87] (i.e. the hash value of the next $q$-gram can be obtained in $O(1)$ from the current one). The occurrences are found in ascending order, hence each insertion takes $O(1)$ time.

Therefore, this index is built in $O(n)$ expected time and a single pass over the text. The worst case can be made $O(n)$ by modifying Ukkonen's technique to build a suffix tree in linear time [Ukk95] (we only want the tree up to height $q$).

#### 9.1.2.2 Index Space

We analyze space now. To determine the number of different $q$-grams in random text, consider that there are $\sigma^q$ different "urns" ($q$-grams) and $n$ "balls" ($q$-grams in the text) to be assigned to the urns. The probability of a $q$-gram to be selected by a ball is $1/\sigma^q$. Therefore, the probability of a $q$-gram *not* being selected in $n$ trials is $(1 - 1/\sigma^q)^n$. Hence, the average number of $q$-grams that appear in the $n$ trials is $V = \sigma^q(1-(1-1/\sigma^q)^n) = \Theta(\sigma^q(1-e^{-n/\sigma^q})) = \Theta(\min(n, \sigma^q))$. This shows that $q$ must be kept $o(\log_\sigma n)$ for the vocabulary space to be sublinear. We show practical sizes in the experiments.

We consider the lists of occurrences now. Since we index all positions of all $q$-grams, the space requirements are $O(n)$, being effectively $4n$ on a 32-bit architecture[2]. If block addressing is used (with blocks of size $b$), we consider that there is an entry in the list of occurrences per different $q$-gram mentioned in each different block. Reusing the analysis of the vocabulary, each block has $\Theta(\min(b, \sigma^q))$ different $q$-grams. Multiplying this by the number of blocks $(n/b)$, we have that the total size of the occurrence lists is $O(n \min(1, \sigma^q/b))$, which is $o(n)$ if and only if $b = \omega(\sigma^q)$o(b), or $q = o(\log_\sigma b)$.

#### 9.1.2.3 Retrieval Time

We now turn our attention to the time to answer a query. The first splitting optimization phase is $O(m^2(k + \log n))$ or just $O(m^2 k)$ as explained. Once we have all the positions to verify, we can check each zone using a classical algorithm at a cost of $O(m^2)$ each. However, we have shown in Section 4.3.1 that the cost of a verification can be made $O(\ell^2)$ on average by using the hierarchical verification technique (where $\ell$ is the length of the searched piece). This cost is exactly the same as in the on-line version of Section 6.1 since it is related to the number of occurrences of the pieces in the text.

We analyze only the case of random text (natural language is shown in the experiments). Under this assumption, we discard the effect of the optimization and assume that the pattern is split in pieces of lengths as similar as possible. In fact, the optimization technique makes more difference in natural language texts, making the approach in that case more similar in performance to the case of random text.

It should be clear that if $k$ and $m$ are fixed, this index can never be sublinear in time, simply because on random text the number of occurrences of the pattern grows linearly as the text grows and we have to verify all those positions. In the analysis which follows we speak in terms of sublinearity and derive order conditions on $b$ and $q$. This is reasonable since we can control them and make them grow as $n$ grows. However, when the conditions imply that $m$ or $k$ must grow as a function of $n$ to obtain the sublinearity, we are in fact meaning that it is not reasonable to consider that such sublinearity is achievable.

We split the pattern in pieces of length $\lfloor m/(k+1) \rfloor$ and $\lceil m/(k+1) \rceil$. In terms of probability of occurrence, the shorter pieces are $\sigma$ times more probable than the others. The total cost of

[2]We store just one pointer for each $q$-gram position. This allows indexing up to 4 Gb of text. Therefore we would use more than four bytes to index longer texts. On the other hand, we are not considering here the possibility of using a compressed list of positions, which can considerably reduce the space requirements, typically to 2 bytes per pointer. Our implementation uses such compressed pointers

verifications is no more than

$$\frac{m^2}{(k+1)\sigma^{\lfloor\frac{m}{k+1}\rfloor}}\, n$$

which is sublinear for $\alpha = o(1/(\log_\sigma(m/\alpha)))$, i.e. $\alpha = o(1/(\log_\sigma m + \log_\sigma \log_\sigma m))$. As this does not involve $q$ or $b$, sublinear verification cost is not achievable.

However, we are not considering that, if $q$ is very small, it is possible that the pieces are longer than $q$. In this case we must truncate the pieces to length $q$ and use the list of occurrences of the resulting $q$-grams. Before triggering a verification on each occurrence of such $q$-grams, we can verify in the text if the occurrence of the $q$-gram is in fact an occurrence of the longer piece. As this takes $O(1)$ time on average for each occurrence of each of the $(k+1)$ lists, we have an additional time of $O(kn/\sigma^q)$, which is sublinear provided $q = \omega(\log_\sigma k)$. This time is obtained by considering that each text character belongs to the $k+1$ selected $q$-grams with probability $O(k/\sigma^q)$.

On the other hand, if we use block addressing, we must find the exact candidate positions before verifying them with the above technique. To do this, we use the on-line algorithm (i.e. that of Section 6.1) which in turn finds the candidate areas and verifies them. Excluding the above considered verifications, the on-line algorithm runs in linear time . Therefore, we show under which restrictions a sublinear part of the text is sequentially traversed. This new condition is stricter than the previous $\alpha = o(1/\log_\sigma(m/\alpha))$.

The probability of a text position matching one piece is, as explained, $(k+1)/\sigma^{\lfloor m/(k+1)\rfloor}$. Therefore, the probability of a block (of size $b$) being sequentially traversed is

$$1 - \left(1 - \frac{k+1}{\sigma^{\lfloor\frac{m}{k+1}\rfloor}}\right)^b$$

and since there are $n/b$ blocks and traversing each one costs $O(b)$, we have that the expected amount of work to traverse blocks is $n$ times the above expression, which is

$$n\left(1 - e^{-\frac{b(k+1)}{\sigma^{\lfloor m/(k+1)\rfloor}}}\right)\left(1 + O\left(k/\sigma^{\lfloor\frac{m}{k+1}\rfloor}\right)\right)$$

and this is sublinear approximately for $b = o(\sigma^{1/\alpha}/m)$

Hence, combined time and space sublinearity is obtained when the block size $b$ satisfies $b = \omega(\sigma^q)$ and $b = o(\sigma^{1/\alpha}/m)$ (we also need $q = \omega(\log_\sigma k)$). Unfortunately, even for $b = 1$ we need $m = o(\sigma^{1/\alpha})$, which is an order condition imposed over variables which are, at least in principle, not dependent on $n$.

### 9.1.3 Experiments

We show experimentally the index construction times and sizes for different values of $q$, with character and block addressing. We also show the querying effectiveness of the indices, by comparing the percentage of the query time using the index against that of using the on-line algorithm. The experimental values agree well with our analysis in terms of the error ratios and block sizes up to where the indices are useful.

For the tests we use a collection of 8.84 Mb of English literary text[3] filtered to lower-case and with all separators converted to a single space. We test the cases $q = 3..5$, as well as character addressing

[3]This includes the writings of Franklin we use in most cases in this thesis, as well as Poe, Emerson, Wilde, Yeats and Andersen.

and block addressing with blocks of size 2 Kb to 64 Kb. Blocks smaller than 2 Kb were of no interest because the index size was the same as with character addressing, and larger than 64 Kb were of no interest because query times were too close to the on-line algorithm.

Figure 9.3 shows index build time and space overhead for different $q$ values and block sizes. The size of the vocabulary file was 61 Kb for $q = 3$, 384 Kb for $q = 4$ and 1.55 Mb for $q = 5$, which shows a sharp increase.

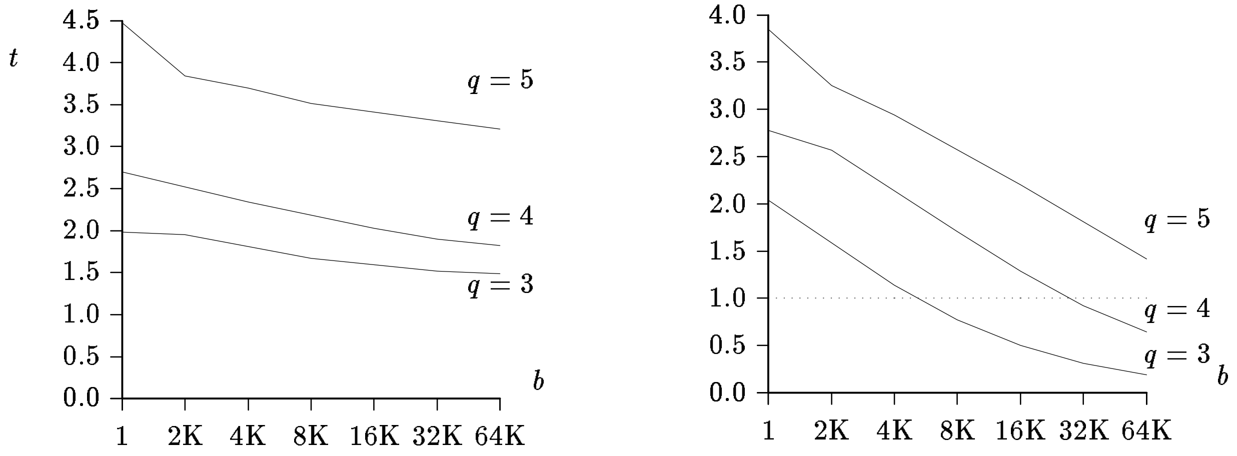


Figure 9.3: On the left, index construction times (minutes of user time). On the right, their space overhead (i.e. index space divided by text space). The dotted line shows a 100% overhead.

We show now query times. We tested queries of length $m = 8$, 16 and 24 (i.e. from a word to a short phrase). The queries were randomly chosen from the text at the beginning of non-stopwords. This setup mimics common text retrieval scenarios. For $m = 8$ we show tests with $k = 1$ and 2; for $m = 16$ with $k = 1..4$ and for $m = 24$ with $k = 1..6$. Every data point was obtained by averaging Unix's user time over 100 random trials.

Figure 9.4 shows the percentage of text traversed by using the index (the on-line algorithm should traverse the whole text). As it can be seen, the percentage of text traversed is very low for the index that stores the exact occurrences of the $q$-grams. The block addressing indices, on the other hand, traverse much more text and they are useful only for small block sizes.

Figure 9.5 shows actual query execution times as a percentage of the on-line algorithm. It can be seen that the situation worsens. This happens because there is an important overhead in manipulating the index. This not only plays against the indexed algorithms, but even makes it better to use the on-line algorithm when the filtration efficiency of the index is not good (moreover, the indices with larger $b$ become better because the overhead is less and the verifications are the same). In the character addressing index, this happens for $\alpha > 1/4$. Up to that point, the search times are under 10 seconds. The block addressing indices, on the other hand, cease to be useful too soon, namely for $\alpha > 1/8$.

Finally, we show the effect of our splitting optimization technique, by comparing, for character addressing indices, the retrieval times using and not using the optimization. As Figure 9.6 shows, the improvement due to the optimization is very significant. Even when the length of the $q$-grams do not allow selecting longer pieces, the optimization technique selects the least frequent $q$-grams.

There is some work about this index that we leave for the future. We should incorporate the hierarchical verification techniques to improve its behavior for intermediate error levels. There are also more possible improvements which are specific for the indexed version. Pattern pieces longer

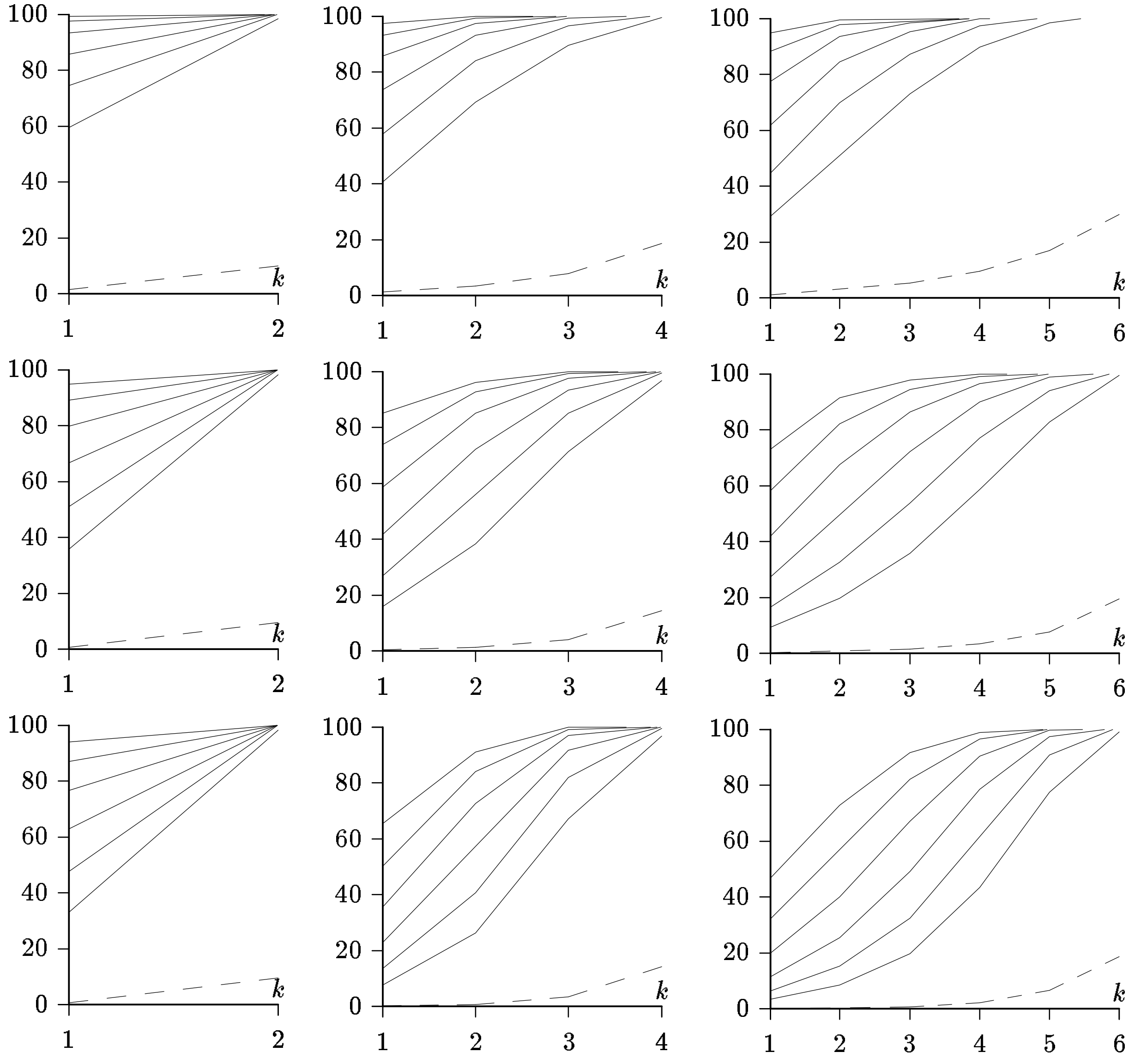


Figure 9.4: Percentage of text traversed using the index. The rows correspond to $q = 3$, 4 and 5, and the columns to $m = 8$, 16 and 24. The dashed line corresponds to character addressing, solid lines to block addressing. From lower to upper they correspond to $b = 2$, 4, 8, 16, 32 and 64 Kb.

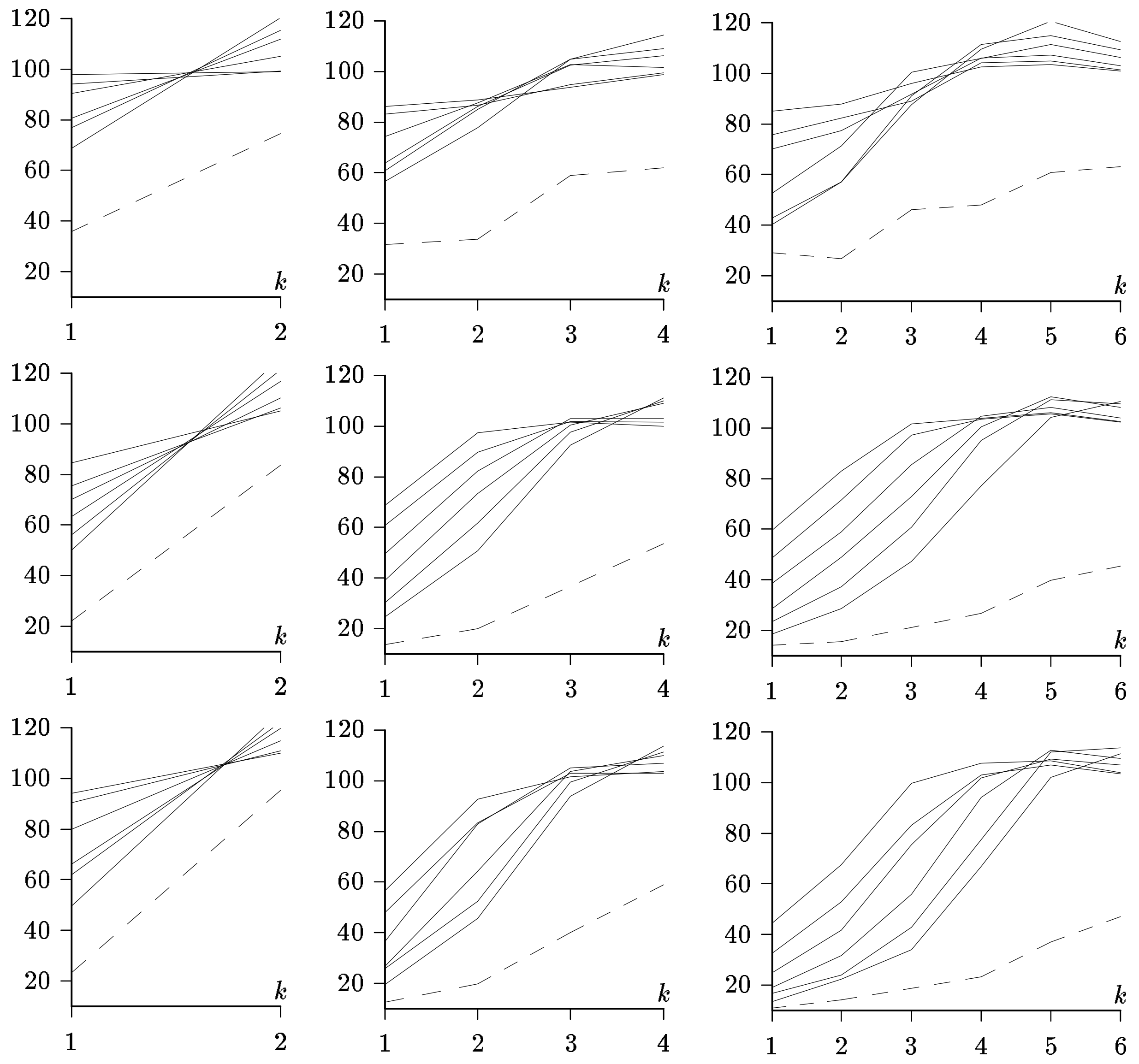


Figure 9.5: Query time using the index divided by query time using the on-line algorithm (percentage). The rows correspond to $q$ = 3, 4 and 5, and the columns to $m$ = 8, 16 and 24. The dashed line corresponds to character addressing, solid lines to block addressing. From lower to upper (at $k = 1$) they correspond to $b$ = 2, 4, 8, 16, 32 and 64 Kb.

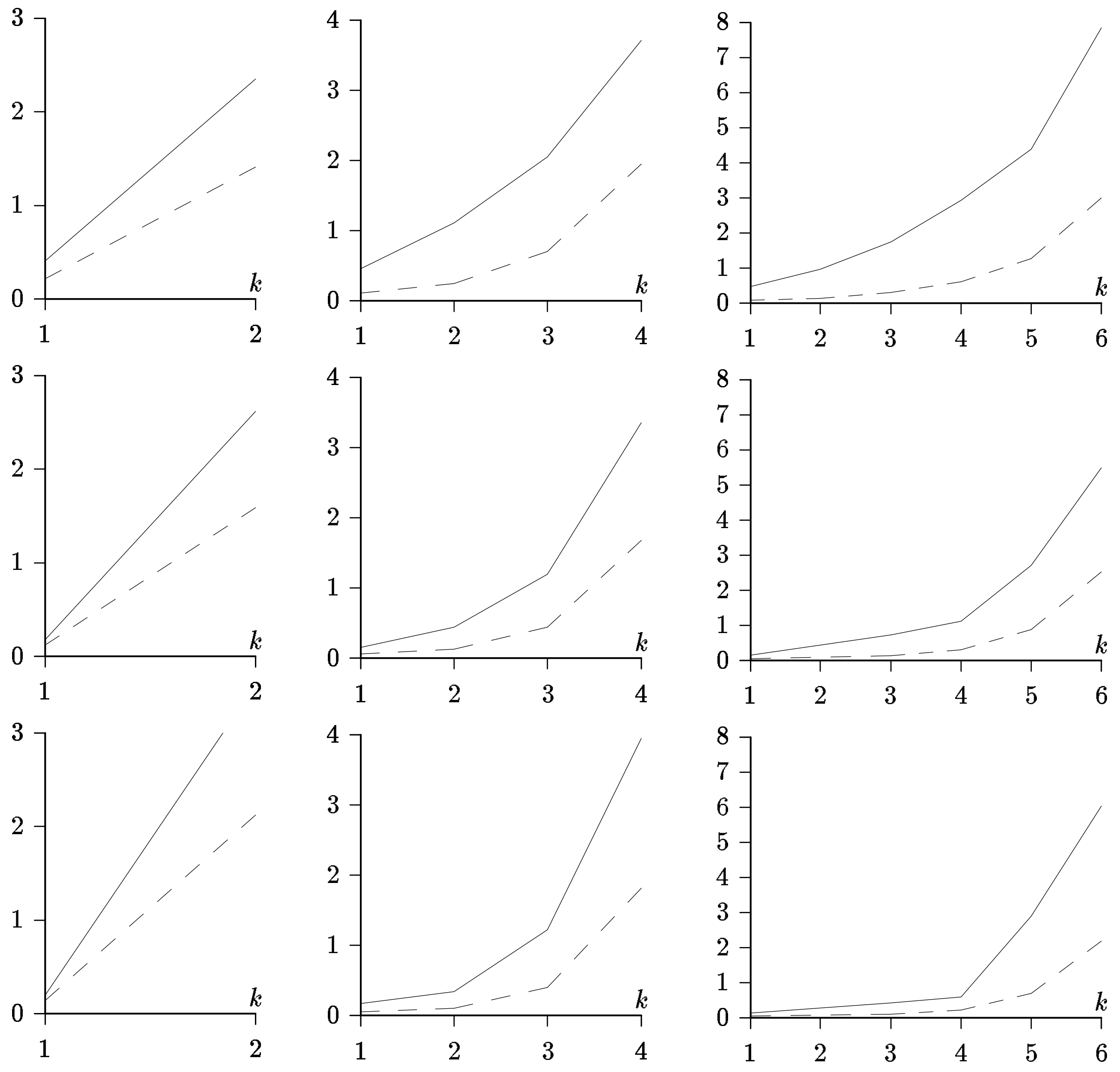


Figure 9.6: Comparison of retrieval times using the splitting optimization technique (dashed line) versus not using it (solid line), for the character addressing index. The rows correspond to $q = 3$, 4 and 5, and the columns to $m = 8$, 16 and 24.

than $q$ are truncated. This loses part of the information on the pattern. This case could justify the approach of [Shi96] of splitting the pattern in more than $k+1$ pieces and forcing more than one piece to match before verifying. Extending the scheme to matching more than one piece reduces the number of verifications but leads to a more complex algorithm, whose costs may outweight the gains of less verifications. Another interesting idea which has not been pursued is to try many splits and to intersect the results (somehow resembling [GPY94]). We are currently studying these issues.

A difficult problem for word-retrieving indices is to account for errors involving separators (since they split the words). On an index like this one this is not a problem, but this index takes more space. However, if the search pattern does not include separators we should not index any $q$-gram including a separator, which drastically reduces their number. Moreover, we could use the text words (like word-retrieving indices) instead of the $q$-grams, and search the pattern pieces inside the words, with a fast on-line exact searching algorithm. This would be in fact our partitioning into exact search implemented over a traditional inverted index. We leave this for future work as well.

The question of sublinearity for sequence-retrieving indices is interesting. In the natural language model we can prove query time sublinearity by assuming that we only search for complete words, and using heuristic rules which show that the vocabulary of a text grows as it grows. On the other hand, sublinearity can never occur if we consider random text, since each new character has a fixed probability of matching and therefore the output is of size $\Omega(n)$. The indices based on suffix trees achieve sublinear time (in fact, independent on $n$) but they do not report all the matches, just subtrees whose leaves contain all the matches. Whether the number of matches on natural language is or not sublinear if we allow any sequence (not only words) to match our query is an interesting subject of future study, as it can settle the question of the possibility of sublinear-time sequence-retrieving indices for natural language text. On the other hand, one can consider that the number of matches is very low and count them in a separate variable $R$. Although formally we have $R = \Omega(n)$, for practical queries the constant is very small.

## 9.2 An Index Based on Suffix Trees

Suffix trees are used as indices for approximate string matching because they factor out the repetitions that occur in the text. As explained in Section 3.3.2, the idea is to simulate the text traversal of an on-line algorithm, running it on the suffix tree instead of on the text. The algorithm which minimizes the number of nodes traversed is [Cob95], while [BYG90, Gon92] is simpler but inspects more nodes. In this section we show that the latter index can be adapted to use a node processing algorithm which is faster than dynamic programming, namely our algorithm of Chapter 5. We give analytical results for this technique, and test it experimentally in the next section. We finish this section by presenting a new technique based on pattern partitioning, so that the pattern is split in many patterns which are searched in the suffix tree and their occurrences verified for a complete match. We show in the experiments of the next section that this technique outperforms all the others.

### 9.2.1 Using the Bit-parallel Automaton

The idea of [BYG90, Gon92] is a limited depth-first search on the suffix tree. Since every substring of the text (i.e. every potential occurrence) starts at the root of the suffix tree, it is sufficient to explore every path starting at the root, descending by every branch up to where it can be seen that that branch does not represent the beginning of an occurrence of the pattern.

More specifically, we compute the edit distance between the tree path and the pattern, and if at some node we obtain a distance of $k$ or less then we know that the text substring represented by the node matches the pattern. We report all those nodes (to obtain the actual matches it is necessary to traverse all the leaves of the suffix tree which descend from those nodes, since they are extensions of the matching substring). On the other hand, when we can determine that the edit distance cannot be as low as $k$, we abandon the path. This happens at depth $m + k + 1$ but can happen before.

We propose now a variation on the previous algorithm, based on our algorithms of Chapter 5. The idea is to replace the use of dynamic programming by our bit-parallel on-line algorithm over the suffix tree. This algorithm uses bit parallelism to simulate an automaton that recognizes the approximate pattern. It improves the performance, achieving linear time for small patterns. If the pattern is long, the automaton is partitioned in many computer words.

We need to modify the automaton to compute edit distance (Section 5.3). Hence, we remove the initial self-loop. We do not need to add the initial lower-left triangle, since if a substring matches with initial deletions we will find (in other branch) another version of it which does not need the deletions (to see this, notice that if a 1 finally exits from the lower-left triangle and reaches the final state, then the same path can be followed by a suffix of the text area that matched). When the automaton runs out of active states we can abandon the search.

This bit-parallel variation is only possible because of the simplicity of the traversal. For instance, the idea does not work on the more complex setup of [Ukk92, Cob95], since these need some adaptations of the dynamic programming algorithm that are not easy to parallelize. Hence, the tradeoff is: we use a faster algorithm to process the nodes, but we cannot use a smart algorithm to traverse less nodes. We show experimentally that this idea pays off.

As we have shown that approximate string matching is simulated using an automaton, this algorithm can be seen as a particular case of general automaton searching over a trie [BYG96]. However, in this case the automaton is nondeterministic and converting it to deterministic is not practical, since it tends to generate large automata (see Section 6.4).

### 9.2.2 Analysis

An asymptotic analysis on the performance of a depth-first search over suffix trees is immediate if we consider that we cannot go deeper than level $m + k$ since past that point the edit distance between the path and our pattern is larger than $k$ and we abandon the search. Therefore, the most we can work is $O(\sigma^{m+k})$, which is independent on $n$ and hence $O(1)$. Another clear way to see this is to use the analysis of [BYG96], where the problem of searching an arbitrary regular expression over a suffix trie is considered. This includes this case, because we run an automaton. Their result for this case indicates constant time (i.e. depending on the size of the automaton only) because the automaton has no cycles.

However, we are interested in a more detailed analysis, namely when $n$ is not so large in comparison to $m + k$. We first analyze which is the average number of nodes at level $\ell$ in the suffix tree of the text, for small $\ell$. Since almost all suffixes of the text are longer than $\ell$ (i.e. all except the last $\ell$), we have nearly $n$ suffixes that reach that level. However, not all these suffixes are different in their first $\ell$ characters. The total number of nodes at level $\ell$ is the number of different suffixes once they are pruned at $\ell$ characters. This is the same as the number of different $\ell$-grams in the text. We can reuse the analysis of Section 9.1.2.2 to find out that the number of such nodes is $\Theta(\sigma^\ell(1 - e^{-n/\sigma^\ell})) = \Theta(\min(n, \sigma^\ell))$ if the text is random. That shows that the average case is close to the worst case: up to level $\log_\sigma n$ all the possible $\sigma^\ell$ nodes exist, while for deeper levels all the different $n$ nodes exist. We thus work on this worst case.

We now get into the second part of the analysis. We need to determine which is the probability of the automaton being active at a given node of depth $\ell$ in the suffix tree. Notice that the automaton is active if and only if some automaton state of the last row is active. This is equivalent to some *prefix* of the pattern matching with $k$ errors or less the text substring represented by the suffix tree node under consideration. So we can partially reuse our analysis of Section 4.1.

Since we are computing edit distance now, the probability of a pattern prefix of length $m'$ matching a text substring of length $\ell$ must consider all the characters of the text substring. As done in Section 4.1, we have that at least $\ell - k$ of the text must match the pattern when $\ell \geq m'$, and that at least $m' - k$ characters of the pattern must match the text whenever $m' \geq \ell$. Hence, the probability of matching is upper bounded by

$$\frac{1}{\sigma^{\ell-k}}\binom{\ell}{\ell-k}\binom{m'}{\ell-k} \qquad \text{or} \qquad \frac{1}{\sigma^{m'-k}}\binom{\ell}{m'-k}\binom{m'}{m'-k}$$

depending on whether $\ell \geq m'$ or $m' \geq \ell$, respectively. Notice that this imposes that $m' - k \leq \ell \leq m' + k$. We also assume $m' \geq k$, since otherwise the matching probability is 1. As $k \leq m' \leq m$, we have that $\ell \leq m + k$, otherwise the matching probability is zero. Hence the matching probability is 1 for $\ell \leq k$ and 0 for $\ell > m + k$, and we are interested in what happens in between.

Since we are interested in any pattern prefix matching the current text substring, we add up all the possible lengths from $k$ to $m$:

$$\sum_{m'=k}^{\ell} \frac{1}{\sigma^{\ell-k}}\binom{\ell}{\ell-k}\binom{m'}{\ell-k} + \sum_{m'=\ell+1}^{m} \frac{1}{\sigma^{m'-k}}\binom{\ell}{m'-k}\binom{m'}{m'-k}$$

which is very similar to Eq. (4.3) if we replace the $s$ and $m$ there by $m'$ and $\ell$ here, respectively. The only differences are the absence of "$-2$" (which does not affect the outcome) and the upper limit of the second summation, which is $m + k$ there and is not $\ell + k$ here. In this formula the limit is $m$, which is not upper bounded in terms of $\ell$ and $k$.

There is in fact a semantic difference which (fortunately) is not reflected in the formula: here we are summing over different pattern lengths and a fixed text length, while in Section 4.1 we summed over different text lengths for a fixed pattern length. This is a consequence of the symmetry of the problem.

All the reasoning of Section 4.1 can be followed in order to obtain an equivalent to Eq. (4.2), where $\alpha = k/\ell$. The only point where the difference in the upper limit plays a role is when we show that $\delta h(x)/\delta x$ does not have roots in the interval of the second summation, but in fact there are no roots after $m' \geq \ell$, and therefore the upper limit is not important.

Hence, the result is that the matching probability is very high for $\alpha = k/\ell > 1 - e/\sqrt{\sigma}$, and otherwise it is $O(\gamma^\ell)$ for $\gamma < 1$. Therefore, we can pessimistically consider that in levels

$$\ell \;\leq\; L(k) \;=\; \frac{k}{1 - e/\sqrt{\sigma}} \;=\; O(k)$$

all the nodes in the suffix tree are visited, while deeper nodes at level $\ell > L(k)$ are visited with probability $O(\gamma^\ell)$ for $\gamma < 1$ (notice that this result is very similar to that of Eq. (4.5)).

We are left with three disjoint cases to analyze, illustrated in Figure 9.7. We say that the index is "useful" if the number of nodes it traverses does not grow proportionally to the text size $n$.

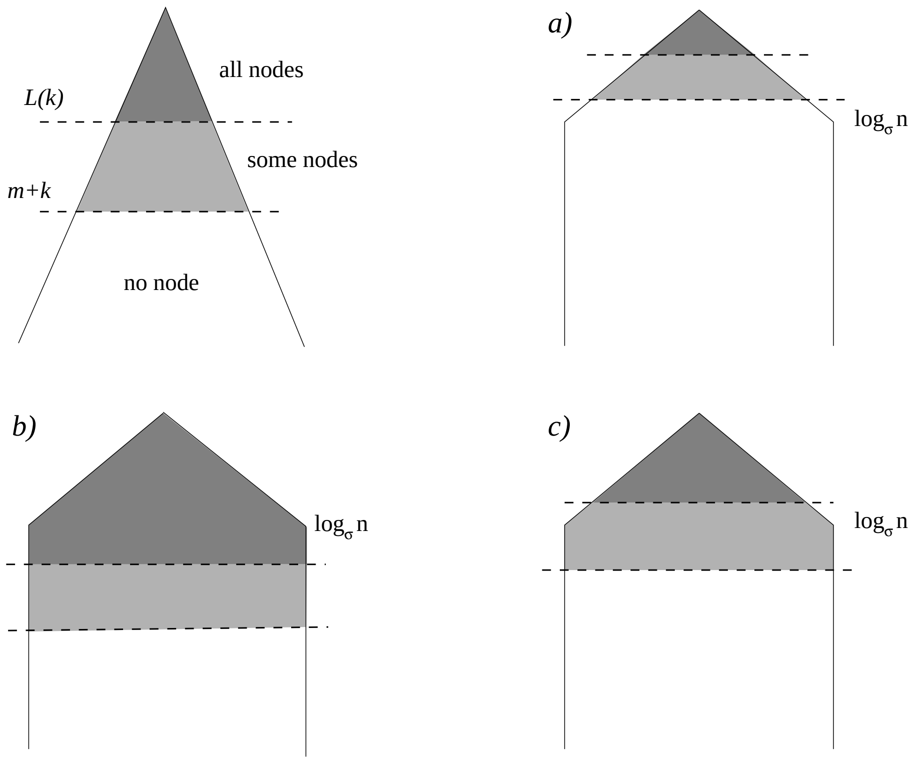


Figure 9.7: The upper left figure shows the visited parts of the tree. The rest shows the three disjoint cases in which the analysis is split.

*a)* $m + k < \log_\sigma n$, i.e. $n > \sigma^{m+k}$ or "very large $n$"

It is clear in this case that, even if we worked on *all* the possible nodes (i.e. up to depth $m + k$), the total amount of work would still be less than $n$ and would not grow as $n$ grows.

Therefore, the index is always useful in this case, whose condition is equivalent to $\alpha < (\log_\sigma n)/m - 1$.

*b)* $L(k) \geq \log_\sigma n$, i.e. $n \leq \sigma^{L(k)}$, or "very small $n$"

In this case, since on average we work on all the nodes up to level $\log_\sigma n$, the total work is $n$, i.e. the amount of work is proportional to the text size. This shows that the index simply does not work for very small texts, being an on-line search preferable.

Therefore, the index is never useful in this case, whose condition is equivalent to $\alpha > (\log_\sigma n)/m \ (1 - e/\sqrt{\sigma})$.

*c)* $L(k) < \log_\sigma n \leq m + k$, i.e. "intermediate $n$"

In this case, we work on all nodes up to $L(k)$ and on some nodes up to $m + k$. The total amount of nodes visited is

$$\sum_{\ell=0}^{L(k)} \sigma^\ell + \sum_{\ell=L(k)+1}^{\log_\sigma(n)-1} \gamma^\ell \sigma^\ell + \sum_{\ell=\log_\sigma n}^{m+k} \gamma^\ell n$$

The first term of the expression is proportional to $\sigma^{L(k)}$, and smaller than $n$ because we are assuming $L(k) < \log_\sigma n$. It does not grow as $n$ grows.

Since due to Eq. (4.1) we have that $\gamma\sigma > 1$, the the second summation is at most[4]

$$(\gamma\sigma)^{\log_\sigma n} = n^{1+\log_\sigma \gamma}$$

which is sublinear in $n$ if and only if $\gamma < 1$.

The third summation, provided $\gamma < 1$, is at most

$$\frac{n\gamma^{\log_\sigma n}}{1-\gamma} = \frac{n^{1+\log_\sigma \gamma}}{1-\gamma}$$

which again is sublinear in $n$ only if $\gamma < 1$.

Therefore, the index is useful in this case when $\gamma < 1$. An equivalent condition is $\alpha < (\log_\sigma n)/m\ (1 - e/\sqrt{\sigma}) \quad \wedge \quad \alpha < 1 - e/\sqrt{\sigma}$.

Since the total time is proportional to the number of nodes visited, we conclude that

- If $(\log_\sigma n)/m \leq 1$, the retrieval time grows sublinearly if and only if $\alpha < (\log_\sigma n)/m\ (1 - e/\sqrt{\sigma})$.
- If $(\log_\sigma n)/m > 1$, the retrieval time grows sublinearly if and only if $\alpha < (\log_\sigma n)/m - 1$ or $\alpha < 1 - e/\sqrt{\sigma}$.

The cost to inspect a node is $O(1)$ for small patterns (i.e. $(m-k)(k+2) \leq w$), while otherwise it is $O(k(m-k)/w)$ (in the original scheme [BYG90, Gon92] the cost to inspect a node is $O(m)$).

Notice that we have not considered the time to report the matches, we just return subtrees of the suffix tree whose leaves are answers. If we had to enumerate all the matches, we could never achieve sublinear time, since if $f(m,k)$ is the matching probability per character, there exist on average $f(m,k)n = \Omega(n)$ text positions to report.

## 9.2.3 A New Algorithm Based on Pattern Partitioning

From the analysis it is clear that we prefer that $m$ and $k$ be small numbers. We present here a new algorithm that partitions the original pattern into smaller sub-patterns, which have to be located and produce a set of potential candidates. Those candidates are later verified using an on-line algorithm on the text occurrences. This idea is based on the pattern partitioning technique of Chapter 5.

The new algorithm follows. We divide the pattern in $j$ pieces, such that each piece can be searched with the simple (non-partitioned) automaton. Then we search in the suffix tree the $j$ pieces using the algorithm we proposed in Section 9.2.1. We then collect all the matches found and verify all them in the text for a complete occurrence.

Since we perform $j$ searches of the same kind of Section 9.2.1, the same analysis holds provided we multiply the cost by $j$ and replace $m$ by $m/j$ and $k$ by $k/j$. Recall that the $j$ given by pattern partitioning is $j = O((m-k)/\sqrt{w})$. However, to achieve search sublinearity, apart from the considerations of Section 9.2.1, we need also that the total number of verifications be sublinear (no verifications are required in Section 9.2.1). This, unfortunately, is not possible since as explained each text position has a fixed probability of matching the pattern, and therefore the total number

[4]In the case of $\gamma\sigma \leq 1$ it would be even less, i.e. at most $\log_\sigma n$.

of matches is $\Omega(n)$. As we need to verify at least all those matches, we pay $\Omega(n)$. Notice that the approach of Section 9.2.1 does not consider the task of reporting the matches, since it reports suffix tree nodes which root all the leaves that are answers.

As we show experimentally in the next section, this idea works by far better than all the other existing indexing schemes. This is because the search on a suffix tree degrades quickly as $m$ or $k$ grow, up to the point where it is better to perform $j$ searches with patterns of length $m/j$ and $k/j$ errors. If we partition the pattern, however, we immediately loose the sublinearity because the number of verifications to perform grows at the same rate of the text size $n$. On the other hand, this is an asymptotic result and in practice the idea works well.

We may consider the problem of which is the best partition to perform. We initially used the $j$ given by pattern partitioning (i.e. the one making the subautomata fit in a computer word), but the experimental results show that we probably would have obtained better results by partitioning even more. This is a subject independent of which type of automaton we use to search in the nodes. The larger $j$ is, the more searches we will have to perform and the more verification work we will do, but those searches themselves will be much cheaper.

In the extreme case, we can partition the pattern in $k+1$ pieces and search them in the suffix tree with zero errors. Searching all the pieces in the suffix tree costs $O(m)$, and later we have to verify all their occurrences. It is not hard to see that this is basically what our index of samples of Section 9.1 does. The difference is that it prunes the suffix tree at depth $q$, and therefore we must in some cases content ourselves with a pruned pattern piece. But, as the experiments show, even in the cases where $q$ is large enough, the performance of the index is not as good as that of pattern partitioning. This shows that we are paying too much to verify the pieces, and that the optimum is not in this extreme (the other extreme is not partitioning at all, which as clearly shown by the experiments is not the optimum either).

Hence, the best choice is in between. We have to balance between traversing too many nodes of the suffix tree and verifying too many text positions. In fact, the other index which does precisely this is Myers' index [Mye94], which is explained in Section 3.3.3. Myers' index collects all text $q$-grams, and given the pattern it generates all the strings at distance at most $k$ from the pattern, searches them in the index and merges the results. This is the same work of a suffix tree provided that we do not enter too deep (i.e. $q \geq m+k$). If $q < m+k$, Myers' approach splits the pattern and searches the subpatterns in the index, checking all the potential occurrences.

There are two fundamental differences between Myers' index and our approach. First, Myers' index is not a suffix tree and therefore it limits the maximum value for $m+k$ beforehand. This is a reasonable way to reduce space requirements, and gives the method to select $j$. Second, it generates all the strings at a given distance and searches them, instead of traversing the structure to see which of them exist. This makes Myers' approach degrade as the alphabet size grows (in the experiments we show that it works well on DNA but very badly on English). It would be much better to simulate the traversal on a suffix tree using the index of substrings. Each movement in the suffix tree is replaced by a binary search on the set of (sorted) substrings, which adds a modest additional $O(\log n)$ factor to the search time.

We believe that indexing substrings and simulating a suffix tree traversal on them is a very interesting choice, which extends naturally our index based on samples, corrects the bad choice of generating all the strings in Myers' index, and allows to have search times similar to our pattern partitioning index without its huge space requirements. We leave the research on this index for future work.

## 9.3 Experimental Results

We compare our indices with the other existing proposals. However, as the task to program an index is rather heavy, we have only considered other indices when they are already implemented. Hence, the indices included in this comparison are

**Myers':** The index proposed by Myers [Mye94], described in Section 3.3.3. The implementation we use is from the author, and it is able to search some lengths only, not an arbitrary $m$. The allowed lengths depend on the alphabet size.

**Cobbs':** The index proposed by Cobbs [Cob95], described in Section 3.3.2. We use the implementation of the author, which is not optimized for space (although the suffix tree index is naturally space-demanding). Actually the search does not use a suffix tree but a DAWG (see Section 2.7), which is similar. The code is restricted to work on an alphabet of size 4 or less.

**Samples($q$):** Our index based on samples presented in Section 9.1. We show the results for $q = 3$ to 6.

**Dfs($a/p$):** Our index based on suffix trees presented in Section 9.2. We show the results for the base technique ($a$) and pattern partitioning ($p$), explained in Sections 9.2.1 and 9.2.3, respectively.

In particular, approximate searching on other $q$-gram indices (see Section 3.3.3) is not yet implemented and therefore is excluded from our tests. We know, however, that their space requirements are very low (close to a word-retrieving index), but also that since the index simulates the on-line algorithm, its tolerance to errors is very low to be practical (see Section 5.9).

All the indices were set to show the matches they found, in order to put them in a reasonably real scenario. We used two different texts for our experiments:

- DNA text ("h.influenzae"), which is a 1.34 Mb file composed solely of the letters {A,C,G,T}. This file is called DNA in our tests, and H-DNA is the first half megabyte of it.
- English literary text (from B. Franklin), which is filtered as explained in Chapter 2 except for line breaks which are also converted to spaces (actually, we use underscores instead of spaces to avoid problems with some of the indices). This text has 1.26 Mb, and is called FRA in the experiments. The text called H-FRA is the first half megabyte of FRA. Observe that Cobbs' index cannot be built on this text because of its restrictions to the alphabet size.

The texts are rather small, in some cases too small to appreciate the speedup obtained with some indices. This is because of the limitations imposed by suffix trees (we had problems to build the suffix trees for the texts larger than half a megabyte). However, the experiments still serve to obtain basic performance numbers on the different indices.

We present the user and system times to build the indices and the space they take in Table 9.1. The first clear result of the experiment is that the space usage of the indices is very high. In particular, the indices based on suffix trees or DAWGs (Dfs and Cobbs') take 35 to 65 times the text size. This outrules them except for very small texts (for instance, building Cobbs' index on 1.34 Mb took 12 hours of real time in our machine of 64 Mb of RAM). From the other indices, Myers' took 7-9 times the text size, which is much better but still too much in practice. The best option in terms of space is our Samples index, which takes from 1 to 7 times the text size, depending on $q$ and $\sigma$. The larger

| Indexer | DNA | H-DNA | FRA | H-FRA |
|---|---|---|---|---|
| Myers' | 5.84u+0.35s<br>10.68 Mb (7.97X) | 2.08u+0.12s<br>4.50 Mb (9.00X) | 5.22u+0.34s<br>9.39 Mb (7.46X) | 2.01u+0.12s<br>4.18 Mb (8.35X) |
| Samples(3) | 3.84u+0.15s<br>1.53 Mb (1.14X) | 1.39u+0.07s<br>0.57 Mb (1.15X) | 9.29u+0.28s<br>2.52 Mb (2.00X) | 3.51u+0.12s<br>1.02 Mb (2.05X) |
| Samples(4) | 5.53u+0.19s<br>2.04 Mb (1.52X) | 1.95u+0.10s<br>0.77 Mb (1.53X) | 15.05u+0.41s<br>3.48 Mb (2.77X) | 5.90u+0.24s<br>1.48 Mb (2.98X) |
| Samples(5) | 7.37u+0.24s<br>2.48 Mb (1.85X) | 2.62u+0.08s<br>0.94 Mb (1.87X) | 20.82u+0.70s<br>5.18 Mb (4.11X) | 8.70s+0.35s<br>2.32 Mb (4.65X) |
| Samples(6) | 10.53u+0.32s<br>2.90 Mb (2.16X) | 3.88u+0.13s<br>1.11 Mb (2.23X) | 32.86u+1.34s<br>7.65 Mb (6.07X) | 13.19u+0.97s<br>3.54 Mb (7.07X) |
| Cobbs' | 108.70u+532.81s<br>87.99 Mb (65.67X) | 30.50u+76.06s<br>32.93 Mb (65.85X) | n/a | n/a |
| Dfs | 30.89u+104.17s<br>52.25 Mb (38.99X) | 6.48u+0.42s<br>19.55 Mb (39.10X) | 28.46u+76.86s<br>44.66 Mb (35.45X) | 6.43u+0.61s<br>17.66 Mb (35.32X) |

Table 9.1: Times (in seconds) to build the indices and their space overhead. The time is separated in the CPU part ("u") and the I/O part ("s"). The space is expressed in megabytes, and also the ratio index/text is shown in the format $r$X, meaning that the index takes $r$ times the text size.

$q$ or $\sigma$, the larger the index. Samples(5), which takes 2-4 times the text size, performs well at query time.

Compared to its size, Myers' index was built very quickly, thanks to the technique of packing many characters in an integer. The Dfs index, on the other hand, was built faster than Cobbs'. Notice that the suffix trees are built very quickly when they fit in RAM (which happens with the versions of half a megabyte of the texts), but for larger texts the construction time is dominated by the I/O, and it takes too much.

We consider now query times. We have tested short and medium-size patterns, searching with 1, 2 and 3 errors the short ones and with 2, 4 and 6 the medium ones. The short patterns were of length 10 for DNA and 8 for English, and the medium ones were of length 20 and 16, respectively (this is because of the restrictions of Myers' index). We selected 1000 random patterns from each file and use the same set for all the $k$ values of that length, and for all the indices. We present in Tables 9.2 and 9.3 the average time per query measured in milliseconds. We include also the time of on-line searching for comparison purposes. We use our on-line software of Chapter 5, which selects the best strategy given the search parameters (the partitioning into exact searching of Section 6.1 is one of its possible strategies).

The results clearly show a number of facts.

- Our strategy Dfs($a$) of using a simpler traversal algorithm on the suffix tree and in return using a faster search algorithm definitely pays off, since our algorithm is 20 to 150 times faster than Cobbs', at the same or less space requirements. Independently of this fact, the suffix tree indices seem to be basically independent on the type of text, but very sensitive to the growth of $m$ or $k$. In fact, the differences between FRAN and DNA are due to the different values of $m$ used. Cobbs' index is never better than on-line searching, but our faster implementation improves over the on-line search for small $m$ and $k$ values. The big problem with this type of

| Indexer | DNA ($m = 10$) | H-DNA ($m = 10$) | FRA ($m = 8$) | H-FRA ($m = 8$) |
|---|---|---|---|---|
| On-line | 1: 131.0u+21.35s<br>2: 152.6u+20.56s<br>3: 188.7u+20.36s | 1: 55.01u+15.24s<br>2: 62.41u+15.48s<br>3: 84.20u+15.33s | 1: 59.74u+17.31s<br>2: 114.8u+20.86s<br>3: 142.2u+20.56s | 1: 29.99u+9.00s<br>2: 52.77u+11.56s<br>3: 60.30u+13.76s |
| Myers' | 1: 38.46u+227.6s<br>2: 149.3u+227.8s<br>3: 1187u+232.1s | 1: 35.00u+116.3s<br>2: 96.10u+117.4s<br>3: 688.0u+117.8s | 1: 420.5u+198.5s<br>2: 2698u+206.2s<br>3: 3179u+205.5s | 1: 185.0u+99.1s<br>2: 1066u+103.1s<br>3: 1261u+109.2s |
| Samples(3) | 1: 655.3u+2207s<br>2: 1836u+4471s<br>3: 5468u+13668s | 1: 243.0u+813.6s<br>2: 718.5u+1702s<br>3: 2079u+5168s | 1: 60.71u+182.4s<br>2: 377.4u+901.6s<br>3: 1410u+2925s | 1: 29.47u+68.92s<br>2: 129.9u+315.0s<br>3: 552.5u+1202s |
| Samples(4) | 1: 235.2u+779.8s<br>2: 1425u+3186s<br>3: 5788u+13750s | 1: 94.52u+290.6s<br>2: 568.0u+1251s<br>3: 2151u+5179s | 1: 44.80u+109.7s<br>2: 379.2u+915.1s<br>3: 1966u+2973s | 1: 22.38u+46.10s<br>2: 142.2u+30.23s<br>3: 819.1u+1156s |
| Samples(5) | 1: 119.7u+308.5s<br>2: 1507u+3287s<br>3: 6870u+13448s | 1: 50.91u+116.6s<br>2: 583.6u+1244s<br>3: 2600u+5097s | 1: 44.74u+102.1s<br>2: 564.8u+845.6s<br>3: 3392u+29.34s | 1: 23.20u+44.47s<br>2: 183.0u+312.8s<br>3: 1297u+11.82s |
| Samples(6) | 1: 118.5u+295.0s<br>2: 1727u+3369s<br>3: 10816u+14147s | 1: 51.59u+113.6s<br>2: 677.3u+1237s<br>3: 4126u+5081s | 1: 53.12u+105.7s<br>2: 782.0u+907.4s<br>3: 5593u+3002s | 1: 25.78u+45.42s<br>2: 254.0u+324.4s<br>3: 2347u+1185s |
| Cobbs' | 1: 110.0u+192.5s<br>2: 588.1u+1989s<br>3: 3370u+14291s | 1: 101.8u+156.0s<br>2: 377.0u+1113s<br>3: 1835u+6060s | n/a | n/a |
| Dfs($a$) | 1: 3.13u+7.81s<br>2: 54.31u+146.8s<br>3: 397.3u+1218s | 1: 1.45u+0.00s<br>2: 22.29u+0.00s<br>3: 152.8u+0.28s | 1: 4.80u+11.41s<br>2: 35.31u+99.50s<br>3: 157.2u+455.0s | 1: 1.88u+0.03s<br>2: 9.77u+0.05s<br>3: 35.65u+0.10s |
| Dfs($p$) | 1: 3.19u+8.47s<br>2: 1.40u+0.00s<br>3: 2.12u+0.01s | 1: 1.51u+0.00s<br>2: 1.34u+0.00s<br>3: 1.66u+0.00s | 1: 4.44u+9.12s<br>2: 32.51u+66.87s<br>3: 146.5u+439.9s | 1: 1.96u+0.00s<br>2: 9.92u+0.00s<br>3: 34.95u+0.02s |

Table 9.2: Query times (in milliseconds) for short patterns and for 1, 2 and 3 errors. The time is separated in the CPU part ("u") and the I/O part ("s").

| Indexer | DNA ($m = 20$) | H-DNA ($m = 20$) | FRA ($m = 16$) | H-FRA ($m = 16$) |
|---|---|---|---|---|
| On-line | 2: 184.6u+22.18s<br>4: 311.4u+21.70s<br>6: 779.2u+21.42s | 2: 75.16u+16.61s<br>4: 116.0u+15.79s<br>6: 297.4u+15.77s | 2: 60.59u+17.56s<br>4: 116.3u+20.83s<br>6: 205.6u+20.58s | 2: 29.91u+9.48s<br>4: 50.71u+14.98s<br>6: 92.36u+13.37s |
| Myers | 2: 123.6u+225.1s<br>4: 1596u+237.7s<br>6: 13149u+266.4s | 2: 68.21u+112.8s<br>4: 651.0u+105.9s<br>6: 5269u+142.2s | 2: 426.1u+203.4s<br>4: 3802u+205.1s<br>6: 5444u+217.0s | 2: 325.0u+106.2s<br>4: 1619u+111.4s<br>6: 2329u+117.1s |
| Samples(3) | 2: 876.7u+2494s<br>4: 3406u+6342s<br>6: 6567u+13264s | 2: 335.4u+937.3s<br>4: 1296u+2355s<br>6: 2711u+5363s | 2: 41.26u+120.6s<br>4: 255.5u+488.7s<br>6: 1682u+2245s | 2: 22.53u+51.65s<br>4: 105.5u+194.2s<br>6: 648.3u+872.5s |
| Samples(4) | 2: 286.3u+768.8s<br>4: 1912u+2568s<br>6: 7093u+13239s | 2: 120.6u+296.6s<br>4: 714.4u+913.6s<br>6: 2820u+5913s | 2: 26.38u+47.77s<br>4: 242.0u+391.0s<br>6: 1972u+2226s | 2: 18.93u+21.69s<br>4: 102.9u+160.2s<br>6: 817.0u+890.0s |
| Samples(5) | 2: 111.4u+287.3s<br>4: 1637u+2141s<br>6: 7861u+12590s | 2: 48.35u+110.1s<br>4: 669.3u+899.1s<br>6: 3169u+5102s | 2: 22.89u+35.40s<br>4: 257.1u+410.9s<br>6: 3049u+2292s | 2: 18.60u+17.05s<br>4: 109.1u+166.2s<br>6: 1176u+892.1s |
| Samples(6) | 2: 57.30u+114.8s<br>4: 1746u+2141s<br>6: 11878u+13320s | 2: 30.53u+46.22s<br>4: 716.6u+882.2s<br>6: 4550u+5098s | 2: 23.60u+31.03s<br>4: 314.6u+388.2s<br>6: 4698u+2328s | 2: 18.03u+17.71s<br>4: 127.2u+162.9s<br>6: 1781u+894.3s |
| Cobbs' | 2: 726.1u+1700s<br>4: ***<br>6: *** | 2: 496.3u+974.0s<br>4: 8060u+14447s<br>6: *** | n/a | n/a |
| Dfs($a$) | 2: 52.07u+177.4s<br>4: 2106u+7889s<br>6: 11341u+40604s | 2: 18.97u+0.13s<br>4: 437.1u+0.10s<br>6: 2516u+0.00s | 2: 28.98u+69.75s<br>4: 431.4u+1282s<br>6: 2136+6433s | 2: 13.08u+0.03s<br>4: 124.3u+0.10s<br>6: 512.6u+0.00s |
| Dfs($p$) | 2: 7.93u+17.15s<br>4: 6.06u+1.66s<br>6: 94.10u+80.52s | 2: 3.52u+0.00s<br>4: 5.42u+0.00s<br>6: 67.01u+0.01s | 2: 9.01u+15.05s<br>4: 46.50u+109.8s<br>6: 269.8u+722.3s | 2: 3.87u+0.01s<br>4: 20.72u+0.00s<br>6: 77.40u+0.00s |

*** One single query took more than 10 minutes.

Table 9.3: Query times (in milliseconds) for moderate-length patterns, and for 2, 4 and 6 errors. The time is separated in the CPU part ("u") and the I/O part ("s").

index is of course the huge space requirements it poses.

- Myers' index behaves well for DNA text but it worsens quickly as the alphabet size grows (i.e. English text). This is because the number of strings at distance $k$ or less from the query grows exponentially with $\sigma$, and the algorithm generates all those strings. For DNA the index is a good alternative, since although it is 3 to 25 times slower than Dfs($a$), it takes 4 times less space. The index also degrades for medium-size patterns, i.e. when it has to perform pattern partitioning. It is better than on-line searching in the case of 10% of error (this could improve on a larger text). Finally (always on DNA) it is better than our Samples index when the pattern is short, but not when pattern partitioning is necessary.

- The Samples index reaches its optimum performance for $q$ between 5 and 6, depending on the case. Unlike Myers', this index improves as $\sigma$ grows, so it works better on English text than on DNA. In DNA it produces a small index (4 times smaller than Myers') but in general has worse search times (except for $m = 20$ and 10% of error, which is also the only case where the index improves over on-line searching and even gets close to Dfs($a$)). The index for $q = 5$ on English text is half the size of Myers' index, and it also obtains good results for medium patterns and low error levels. As we have shown in Section 9.1, these figures improve for larger texts.

- However, the definitive winner is Dfs($p$), which works on the same data structure of Dfs($a$) but partitions long patterns into many subsearches of short patterns. The query times are by far the lowest among all the indices. The experiments show a non-monotonic behavior as $k$ grows, which is due to different partitioning techniques. For instance, for $m = 10$ and $k = 1$ the search fits in a computer word and hence there is no partitioning, for a total cost of 3.19 on DNA. When we move to $k = 2$ the automaton does not fit in a computer word and two searches with $m = 5$ and $k = 1$ are performed, the time being 1.40 in this case. This shows that the best partitioning has little to do with the shape of the automaton, and that the technique deserves more study, in order to determine the best partitioning and to find a good alternative to replace the suffix tree data structure.

We finish this chapter with Figure 9.8, which illustrates the empirical results on space and query time complexity for the different indices. There is a different plot for each value of $n$, $m$, $k$ and $\sigma$, so we have decided to illustrate the case of short patterns and $k = 1$, on both DNA and English text. The Figure gives a quick grasp of the current time and space tradeoffs.

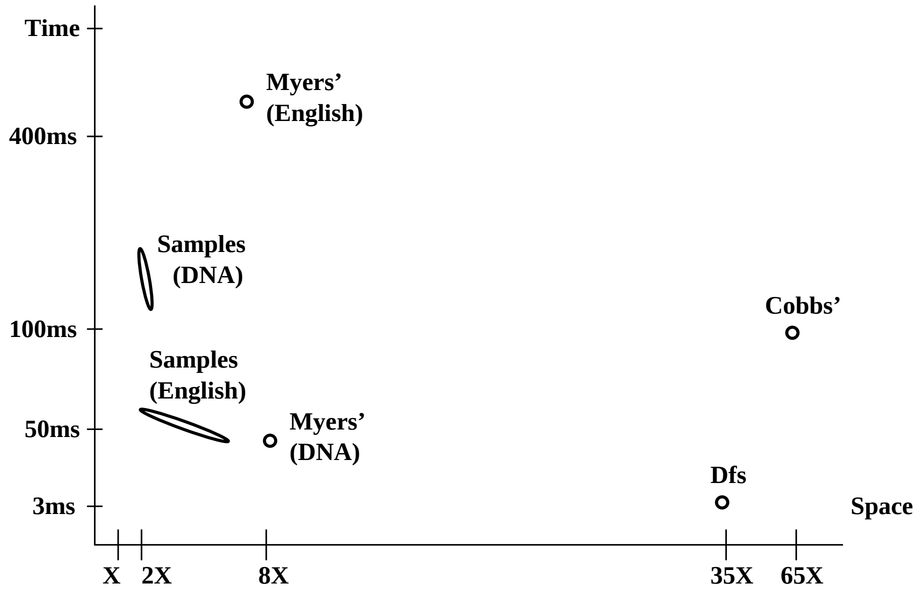


Figure 9.8: Space-time trade-offs on sequence-retrieving indices, in the case of short patterns and one error, and the texts DNA and FRA.

# Chapter 10

# Conclusions

The aim of this thesis was to address the problem of string matching allowing errors, a rather old problem in string matching which is gaining importance in the last years thanks to its applications in computational biology and information retrieval. In this section we present the "big picture" of what has been obtained and what is left.

## 10.1 Results Obtained

We divided the thesis in two main areas: on-line and indexed searching.

On-line searching is the oldest flavor of the problem, and at the same time still the most fast-moving and competitive area. It consists on searching the pattern in the text when there is no possibility to preprocess the text. Our most important achievements in this area are

- We have obtained new results on the probabilistic behavior of approximate searching, proving that the probability of an approximate occurrence is a function of the error level $\alpha$ which goes abruptly from almost zero to almost one, and obtaining theoretical and empirical values for that threshold $\alpha$ value. This allows to understand much better the expected behavior of many search algorithms and to determine which one to use in each case. Despite that some more refined work can still be done, our result is generally enough for most practical purposes.
- We have designed a number of new algorithms, some of which are currently the fastest known on-line search algorithms. In particular, our new algorithms are now the best ones for short patterns and for moderate error levels, which covers almost all the cases of interest in typical text searching.
- We have presented many new algorithms for multipattern approximate searching, which is a rather new area where we expect more movement in the near future. We basically extended many of our algorithms for one pattern to the multipattern case. Currently our algorithms are the best everywhere, except for the case of one error and hundreds of patterns.
- These results are due to strong and novel algorithmic ideas (such as the bit-parallel simulation across diagonals of the NFA, pattern partitioning, hierarchical verification, and superimposition), as well as careful algorithm engineering and successful application and combination of practical techniques (some of them previously known).

Indexed searching, on the other hand, is a newer area (practically born in 1992) where much more development is necessary. In this case the text can be preprocessed and a persistent data structure on it (index) can be built to speed up later querying. Our main achievements for this area are

- We have analytically studied the behavior of word-retrieving indices (those able to retrieve whole words that match the query). One of the most outstanding results is that it is possible to use block addressing to have indices which are sublinear in space overhead and retrieval time simultaneously. This makes an excellent case for the use of this type of index. The result holds also for exact searching. We also analyzed full inverted indices and Web indices.

- We have presented new indexing schemes for sequence-retrieving (i.e. general) indices, which are currently among the best ones. An index based on samples proves to be very adequate for large natural language databases. A second index that combines suffix trees with pattern partitioning seems to be a very promising alternative that could sweep out all the other choices if implemented on a more space-economical data structure. This last index requires further study in order to understand and optimize its behavior.

- We also presented a number of minor tricks which however give impressive improvements on the practical behavior of indices for approximate searching.

As a whole, we believe that this thesis makes a valuable contribution to the field. First, the probabilistic behavior of the problem is much better understood now, together with the expected behavior of on-line and indexed algorithms which depend on these probabilities. Second, we have introduced a number of strong and original algorithmic ideas which have not only been fruitful for us, but could be exploited in the future as well, by ourselves and by other researchers. Third, using these novel ideas and cleverly applying and combining old results and practical techniques, we have obtained new indexing and searching algorithms which are currently the fastest ones in a wide spectrum of parameters of the problem, especially in the areas of typical text searching. One of the most basic lessons learned is that theory and practice must go together in order to achieve the best results, we need good ideas but also simplicity, we need good algorithms but also clever implementations, we need analytical results but also experiments.

## 10.2 Future Work

A number of directions for future work have been mentioned along this thesis at the appropriate points. We collect them here, although we give more details when they are mentioned in their original context.

- Despite that our analysis of approximate matching probability was precise enough for our purposes, a more exact (and still useful) analysis would be interesting.

- We have studied in depth the range of parameters typical of text searching, but other cases with applications to other areas such as small alphabets or very long patterns are interesting as well.

- We believe that there could be still place for improvement on the techniques that try to skip characters in an approximate search. We have presented some new algorithms, and perhaps there are others still waiting.

- For information retrieval, it is interesting to work more on extended patterns and regular expressions combined with approximate searching. We believe that the full power of bit-parallelism has not yet been exploited.

- Working on more complex distance functions is of interest. For instance, few works allow transpositions, despite that they are extremely frequent as typing errors. Under the edit distance, two errors are necessary to simulate a transposition.

- Improving the partitioning into exact search algorithm for natural language could yield interesting practical improvements. For instance, we still need to study more in depth the cost function we minimize for splitting optimization.

- It would be of theoretical interest to analytically or empirically study the growth of the DFA for approximate searching, and of some practical interest to study its behavior with limited memory.

- Improved techniques to handle multiple patterns are yet to be devised. With respect to our techniques, we still need to study better heuristics to group and align the subpatterns.

- We can still optimize more our improved algorithms for word-retrieving indices, looking for better metric space data structures for the vocabulary search or performing a different multi-pattern search per block with only the words that exist in that block.

- It is interesting to join block addressing indices with compression techniques, which drastically reduce the space requirements and give improved search times.

- Not allowing errors involving a separator is the weak point of word retrieving indices. We plan to address this issue by implementing a partition into exact searching over a word retrieving index.

- Our index based on text substrings can be improved in many ways, the most immediate of them being to include hierarchical verification. Other alternatives, such as partitioning in more than $k+1$ pieces or intersecting many different splits, should be studied as well.

- The idea of pattern partitioning on suffix tree indices seems very promising and deserves more study to understand exactly how the process works and which is the best partitioning scheme. We plan to simulate this algorithm on a $q$-gram index to reduce space requirements. Hierarchical verification should also be added here.

## 10.3 Open Questions

There are some interesting aspects in the future of this problem as we foresee it. An important question is: will approximate string matching remain as an interesting problem? We positively believe that the answer is "yes", since most of its sources are not short-term situations. Signals transmitted by physical means will remain having transmission errors (especially if the air becomes a common medium, as the wireless communications gurus hope), and future work on speech recognition will doubtlessly trigger more requirements on this area. DNA and protein analysis will keep being a very active area of research in the foreseeable future (and perhaps even more active than today). Although OCR softwares may improve in the future (we all hope so!), somebody will have to type the texts for the first time (in paper or directly in electronic form), and that people will keep making typing and spelling errors (or alternatively text editing softwares will prevent the errors, in which case those softwares will become new applications for our problem). If we consider that more and more people are getting access to electronic publishing (e.g. news, Web pages, etc.), then it is clear not only that the text databases will keep growing and growing, but also that at least a large proportion of them will be heterogeneous, poorly organized and carelessly written. Searching allowing errors will become the standard practice, and it will be more and more important to find the needle in the haystack of available text. The expected growth of automated linguistic tools to analyze text semantics will probably be another area of application for approximate string matching.

The other question is: is there room for new developments, or is this area closed with respect to what can be done? With respect to on-line searching, it is our belief that it will be very difficult to obtain better results. Bit-parallelism has been exploited to its limit, filtering approaches work so little per text character that hardly could one do less without losing matches. We believe that there is still some room for improvements on techniques that skip characters, but that will not be a breakthrough. Another area which is still open is that of new distance functions, such as to allow transpositions or even more complex distance functions, where many applications in computational biology are waiting. There is also little development for multipattern matching of hundreds of patterns, which will probably become an active area in the future, with applications to spelling and computational linguistics.

There are some theoretical questions still open, such as which is the probability of an approximate match, how does the matching probability behave on natural language or which is the worst-case complexity of this problem if the space is polynomial in $m$.

Other more exotic areas we have not included in this thesis will probably be important in the future. Approximate searching on multidimensional text, for instance, could become a competitor of the current image processing techniques for pattern recognition. Currently, this area needs not only better algorithms, but also to account for rotations and scalings before we can say that it is of real use for subimage searching (there are separate developments for rotations, scaling and errors, but they have not been merged). Approximate string matching on compressed text will be another important area of development, if the tendency of joining textual databases and compression is confirmed. Although there are good solutions for a restricted problem (approximate word matching on natural language text), the general problem is still open.

With regard to indexed approximate string matching, we believe that it is going to be the star problem of this area in the future, not only because indices are the only answer to handle the huge texts that will become commonplace in future information retrieval applications, but also because of its difficulty. When words are matched to words, the problem is rather well solved already and we only expect marginal improvements to the general technique in the future (although the problem of including separators must be solved). On the other hand, if we insist in the general problem, indices are very immature and finding a major improvement would be a real breakthrough. Finding a good index for unrestricted approximate string matching is considered "El Dorado" of this area. Perhaps it exists, perhaps not. Who can tell? We are the explorers...

# Bibliography

[ABF96] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: pattern matching in z-compressed files. *J. of Computer and Systems Sciences*, 52(2):299–307, 1996.

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J. on Computing*, 16:1039–1051, 1987.

[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, June 1975.

[ADKF75] V. Arlazarov, E. Dinic, M. Konrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1975. Original in Russian in *Doklady Akademi Nauk SSSR*, v. 194, 1970.

[AG85] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words.* Springer-Verlag, New York, 1985.

[AG87] A. Apostolico and C. Guerra. The Longest Common Subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[ANZ97] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.

[ANZ98] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. Journal version of [ANZ97], in preparation, 1998.

[Apo85] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

[AS72] M. Abramowitz and I. Stegun, editors. *Handbook of Mathematical Functions.* Dover Publications, NY, 1972.

[BBH$^+$85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The samllest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, New Jersey, 1990.

[BK73] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Comm. of the ACM*, 16(4):230–236, 1973.

[Bla60] C. Blair. A program for correcting spelling errors. *Information and Control*, 3:60–67, 1960.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20(10):762–772, 1977.

[BMNM$^+$93] T. Bell, A. Moffat, C. Nevill-Manning, I. Witten, and J. Zobel. Data compression in full-text retrieval systems. *J. of the American Society for Information Science*, 44:508–531, 1993.

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proc. VLDB'95*, pages 574–584. Morgan Kaufmann, 1995.

[BSSU74] W. Beyer, M. Stein, T. Smith, and S. Ulam. A molecular sequence metric and evolutionary trees. *Mathematical Biosciences*, 19:9–25, 1974.

[BY89] R. Baeza-Yates. *Efficient Text Searching.* PhD thesis, Dept. of Computer Science, Univ. of Waterloo, May 1989. Also as Research Report CS-89-17.

[BY91] R. Baeza-Yates. Some new results on approximate string matching. In *Workshop on Data Structures*, Dagstuhl, Germany, November 1991. (abstract).

[BY92] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.

[BY96] R. Baeza-Yates. A unified view of string matching algorithms. In *SOFSEM'96: Theory and Practice of Informatics*, number 1175 in LNCS, pages 1–15. Springer-Verlag, 1996. Invited paper.

[BYCMW94] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. CPM'94*, number 807 in LNCS, pages 198–212. Springer-Verlag, 1994.

[BYG90] R. Baeza-Yates and G. Gonnet. All-against-all sequence matching. Dept. of Computer Science, University of Chile, 1990.

[BYG92] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Comm. of the ACM*, 35(10):74–82, October 1992. Preliminary version in *SIGIR'89*, 1989.

[BYG94] R. Baeza-Yates and G. Gonnet. Fast string matching with mismatches. *Information and Computation*, 108(2):187–199, 1994. Preliminary version as Tech. Report CS-88-36, Data Structuring Group, Univ. of Waterloo, Sept. 1988.

[BYG96] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on a trie. *J. of the ACM*, 43, 1996.

[BYN96a] R. Baeza-Yates and G. Navarro. A fast heuristic for approximate string matching. In *Proc. WSP'96*, pages 47–63. Carleton University Press, 1996.

[BYN96b] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, number 1075 in LNCS, pages 1–23. Springer-Verlag, 1996.

[BYN97a] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In *Proc. ACM CIKM'97*, pages 1–8. ACM Press, 1997.

[BYN97b] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. WADS'97*, number 1272 in LNCS, pages 174–184. Springer-Verlag, 1997.

[BYN97c] R. Baeza-Yates and G. Navarro. A practical index for text retrieval allowing errors. In R. Monge, editor, *Proc. of the XXIII Latin American Conference on Informatics (CLEI'97)*, pages 273–282, 1997.

[BYN98a] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. Technical Report TR/DCC-98-11, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-blkaddr.ps.gz`.

[BYN98b] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. SPIRE'98*, pages 14–22. IEEE Computer Press, 1998.

[BYN98c] R. Baeza-Yates and G. Navarro. Fast two-dimensional approximate pattern matching. In *Proc. LATIN'98*, number 1380 in LNCS, pages 341–351. Springer-Verlag, 1998.

[BYN98d] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 1998. To appear.

[BYN98e] R. Baeza-Yates and G. Navarro. New and faster filters for multiple approximate string matching. Technical Report TR/DCC-98-10, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/multi.ps.gz`.

[BYNST97] R. Baeza-Yates, G. Navarro, E. Sutinen, and J. Tarhio. Indexing methods for approximate text retrieval. Technical Report TR/DCC-97-2, Dept. of CS, Univ. of Chile, March 1997. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/seqidx.ps.gz`.

[BYP96] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. *Information Processing Letters*, 59:21–27, 1996. Preliminary version in *CPM'92*, LNCS 644, 1992.

[BYR90] R. Baeza-Yates and M. Régnier. Fast algorithms for two dimensional and multiple pattern matching. In *Proc. SWAT'90*, number 447 in LNCS, pages 332–347. Springer-Verlag, 1990.

[BYR98] R. Baeza-Yates and B. Ribeiro, editors. *Modern Information Retrieval.* Addison-Wesley, 1998. To appear.

[CB96] M. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: Evidence and possible causes. In *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 160–169. ACM Press, 1996.

[CCG$^+$94] A. Czumaj, M. Crochemore, L. Gasieniec, S. Jarominek, Thierry Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.

[CH98] R. Cole and R. Hariharan. Approximate string matching: a simpler faster algorithm. In *Proc. ACM-SIAM SODA'98*, pages 463–472. SIAM Press, 1998.

[CL92] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, number 644 in LNCS, pages 172–181. Springer-Verlag, 1992.

[CL94] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12(4/5):327–344, Oct/Nov 1994. Preliminary version in *FOCS'90*, 1990.

[CLR91] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1991.

[CM94] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, number 807 in LNCS, pages 259–273. Springer-Verlag, 1994.

[Cob95] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, Oxford, UK, 1994.

[Cro86] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[CW79] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. ICALP'79*, number 6 in LNCS, pages 118–132. Springer-Verlag, 1979.

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171–176, 1964.

[DFG$^+$97] G. Das, R. Fleisher, L. Gasieniek, D. Gunopulos, and J. Kärkäinen. Episode matching. In *Proc. CPM'97*, number 1264 in LNCS, pages 12–27. Springer-Verlag, 1997.

[DM79] R. Dixon and T. Martin, editors. *Automatic speech and speaker recognition.* IEEE Press, 1979.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[FL95] C. Faloutsos and K. Lin. Fastmap: a fast algorithm for indexing, data mining and visualization of traditional and multimedia datasets. *ACM SIGMOD Record*, 24(2):163–174, 1995.

[FP74] M. Fisher and M. Paterson. String matching and other products. *Complexity of Computation - SIAM AMS Proc.*, 7:113–125, 1974.

[GBY91] G. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, 2nd edition, 1991.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *J. of Complexity*, 4:33–72, 1988.

[GK82] W. Goad and M. Kanehisa. Pattern recognition in nucleic acid sequences. I - A general method for finding local homologies and symmetries. *Nucleic Acid Research*, 10(1):247–263, 1982.

[GKHO97] R. Giegerich, S. Kurtz, F. Hischke, and E. Ohlebusch. A general technique to improve filter algorithms for approximate string matching. In *Proc. WSP'97*, pages 38–52. Carleton University Press, 1997. Preliminary version as Technical Report 96-01, Universität Bielefeld, Germany, 1996.

[GL89] R. Grossi and F. Luccio. Simple and efficient string matching with $k$ mismatches. *Information Processing Letters*, 33(3):113–120, November 1989.

[Gon92] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.

[GP90] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990. Preliminary version in *ICALP'89*, LNCS 372, 1989.

[GPY94] D. Greene, M. Parnas, and F. Yao. Multi-index hashing for information retrieval. In *Proc. FOCS'94*, pages 722–731. IEEE Press, 1994.

[Han93] C. Hancart. On Simon's string searching algorithm. *Information Processing Letters*, 47:95–99, 1993.

[Har95] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[HD80] P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, 1980.

[Hea78] J. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, NY, 1978.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

[HS94] N. Holsti and E. Sutinen. Approximate string matching using $q$-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32. University of Joensuu, 1994.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[JTU96] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Experience*, 26(12):1439–1458, 1996. Preliminary version in Technical Report A-1991-7, Dept. of Computer Science, Univ. of Helsinki, 1991.

[JU91] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, volume 16, pages 240–248. Springer-Verlag, 1991.

[KG82] M. Kanehisa and W. Goad. Pattern recognition in nucleic acid sequences. II - An efficient method for finding locally stable secondary structures. *Nucleic Acid Research*, 10(1):265–277, 1982.

[KMP77] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(1):323–350, 1977.

[Knu73] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Research and Development*, 31(2):249–260, March 1987.

[KS95] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for the inversion distance between two permutations. *Algorithmica*, 13:180–210, 1995.

[KS96] J. Kärkkäinen and E. Sutinen. Lempel-Ziv index for $q$-grams. In *Proc. ESA'96*, number 1136 in LNCS, pages 378–391. Springer-Verlag, 1996.

[KU96] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. WSP'96*, pages 141–155. Carleton University Press, 1996.

[Kur96] S. Kurtz. Approximate string searching under weighted edit distance. In *Proc. WSP'96*, pages 156–170. Carleton University Press, 1996.

[Lev65] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original in Russian in *Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.*

[LMS98] G. Landau, E. Myers, and J. Schmidt. Incremental string comparison. *SIAM J. on Computing*, 27(3):557–582, 1998.

[LS97] T. Luczak and W. Szpankowski. A suboptimal lossy data compression based on approximate pattern matching. *IEEE Trans. on Information Theory*, 43:1439–1451, 1997.

[LST96] O. Lehtinen, E. Sutinen, and J. Tarhio. Experiments on block indexing. In *Proc. WSP'96*, pages 183–193. Carleton University Press, 1996.

[LV88] G. Landau and U. Vishkin. Fast string matching with $k$ differences. *J. of Computer and Systems Science*, 37:63–78, 1988. Preliminary version in *FOCS'85*, 1985.

[LV89] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. of Algorithms*, 10:157–169, 1989. Preliminary version in *ACM STOC'86*, 1986.

[LW75] R. Lowrance and R. Wagner. An extension of the string-to-string correction problem. *J. of the ACM*, 22:177–183, 1975.

[Man52] B. Mandelbrot. An informational theory of the statistical structure of language. In *Proc. Symposium on Applications of Communication Theory*, pages 486–500, 1952.

[Mas27] H. Masters. A study of spelling errors. *Univ. of Iowa Studies in Education*, 4(4), 1927.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23(2):262–272, Apr 1976.

[Mel96] B. Melichar. String matching with $k$ differences by finite automata. In *Proc. ICPR'96*, pages 256–260. IEEE CS Press, 1996. Preliminary version in *Computer Analysis of Images and Patterns*, LNCS 970, 1995.

[MM89] G. Myers and W. Miller. Approximate matching of regular expressions. *Bull. Math. Bio.*, 51:5–37, 1989.

[MM90] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proc. ACM-SIAM SODA'90*, pages 319–327. SIAM Press, 1990.

[MM96] R. Muth and U. Manber. Approximate multiple string search. In *Proc. CPM'96*, number 1075 in LNCS, pages 75–86. Springer-Verlag, 1996.

[MNF57] G. Miller, E. Newman, and E. Friedman. Some effects of intermittent silence. *American J. of Psychology*, 70:311–312, 1957.

[MNF58] G. Miller, E. Newman, and E. Friedman. Length-frequency statistics for written English. *Information and Control*, 1:370–380, 1958.

[MNZBY98a] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. of the 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 90–95. IEEE CS Press, 1998. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/spire98.3.ps.gz`.

[MNZBY98b] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In B. Croft, A. Moffat, C. Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proc. SIGIR'98*, pages 298–306. York Press, 1998.

[Mor68] D. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *J. of the ACM*, 15(4):514–534, October 1968.

[MP80] W. Masek and M. Paterson. A faster algorithm for computing string edit distances. *J. of Computer and System Sciences*, 20:18–31, 1980.

[MW94] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32. USENIX Association, Berkeley, CA, USA, Winter 1994. Preliminary version as Technical Report 93-34, Dept. of Computer Science, Univ. of Arizona, Oct. 1993.

[Mye86a] G. Myers. Incremental alignment algorithms and their applications. Technical Report 86–22, Dept. of Computer Science, Univ. of Arizona, 1986.

[Mye86b] G. Myers. An $O(ND)$ difference algorithm and its variants. *Algorithmica*, 1:251–266, 1986.

[Mye94] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994. Perliminary version in Technical Report TR90-25, Computer Science Dept., Univ. of Arizona, Sept. 1991.

[Mye98] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, number 1448 in LNCS, pages 1–13. Springer-Verlag, 1998.

[Nav97a] G. Navarro. Multiple approximate string matching by counting. In *Proc. WSP'97*, pages 125–139. Carleton University Press, 1997.

[Nav97b] G. Navarro. A partial deterministic automaton for approximate string matching. In *Proc. WSP'97*, pages 112–124. Carleton University Press, 1997.

[Nav98a] G. Navarro. Improved approximate pattern matching on hypertext. In *Proc. LATIN'98*, number 1380 in LNCS, pages 351–357. Springer-Verlag, 1998.

[Nav98b] G. Navarro. Improved approximate pattern matching on hypertext. Technical Report TR/DCC-98-8, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/hyper.ps.gz`.

[NBY98a] G. Navarro and R. Baeza-Yates. Fast multi-dimensional approximate pattern matching. Technical Report TR/DCC-98-7, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-multidim.ps.gz`.

[NBY98b] G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. Technical Report TR/DCC-98-5, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-dexp.ps.gz`.

[NBY98c] G. Navarro and R. Baeza-Yates. A practical index for text retrieval allowing errors. Technical Report TR/DCC-98-9, Dept. of CS, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/practidx.ps.gz`.

[NBY98d] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. Technical Report TR/DCC-98-6, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/hpexact.ps.gz`.

[Nes86] J. Nesbit. The accuracy of approximate string matching algorithms. *J. of Computer-Based Instruction*, 13(3):80–83, 1986.

[NR98a] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. CPM'98*, number 1448 in LNCS, pages 14–33. Springer-Verlag, 1998.

[NR98b] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. Technical Report TR/DCC-98-4, Dept. of Computer Science, Univ. of Chile, 1998. Submitted. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/bndm2.ps.gz`.

[NR98c] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. Technical Report TR/DCC-98-12, Dept. of Computer Science, Univ. of Chile, 1998. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-lzsrch.ps.gz`.

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.

[OM88] O. Owolabi and R. McGregor. Fast approximate string matching. *Software Practice and Experience*, 18(4):387–393, 1988.

[PW95] P. Pevzner and M. Waterman. Open combinatorial problems in computational molecular biology. In *Proc. 3rd Israel Symposium on Theory of Computing and Systems*, pages 158–163. IEEE Press, 1995.

[Rib97] V. Ribeiro. Personal communication. 1997.

[Riv76] R. Rivest. Partial-match retrieval algorithms. *SIAM J. on Computing*, 5(1), 1976.

[RS97] M. Régnier and W. Szpankowski. On the approximate pattern occurrence in a text. In *Proc. Compression and Complexity of SEQUENCES'97*. IEEE Press, 1997.

[San72] D. Sankoff. Matching sequences under deletion/insertion constraints. In *Proc. of the National Academy of Sciences of the USA*, volume 69, pages 4–6, 1972.

[Sel80] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[SF96] R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, 1996.

[Sha77] M. Shapiro. The choice of reference points in best-match file searching. *Comm. of the ACM*, 20(5):339–343, 1977.

[Shi96] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.

[Sim94] I. Simon. String matching algorithms and automata. In *Results and Trends in Theoretical Computer Science*, number 814 in LNCS, pages 386–395. Springer-Verlag, 1994.

[SK83] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[SM96] H. Shang and T. Merrettal. Tries for approximate string matching. *IEEE Transactions on Knowledge and Data Engineering*, 8(4), August 1996.

[ST95] E. Sutinen and J. Tarhio. On using $q$-gram locations in approximate string matching. In *Proc. ESA'95*, number 979 in LNCS, pages 327–340. Springer-Verlag, 1995.

[ST96] E. Sutinen and J. Tarhio. Filtration with $q$-samples in approximate string matching. In *Proc. CPM'96*, number 1075 in LNCS, pages 50–61. Springer-Verlag, 1996.

[Sun90] D. Sunday. A very fast substring search algorithm. *Comm. of the ACM*, 33(8):132–142, August 1990.

[SV97] S. Sahinalp and U. Vishkin. Approximate pattern matching using locally consistent parsing. Manuscript, University of Maryland Institute for Advanced Computer Studies (UMIACS), 1997.

[Tak94] T. Takaoka. Approximate pattern matching with samples. In *Proc. ISAAC'94*, number 834 in LNCS, pages 234–242. Springer-Verlag, 1994.

[TU93] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM J. on Computing*, 22(2):243–260, 1993. Preliminary version in *SWAT'90*, LNCS 447, 1990.

[Uhl91] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[Ukk85a] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985. Preliminary version in *Proc. Int. Conf. Found. Comp. Theory*, LNCS 158, 1983.

[Ukk85b] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[Ukk92] E. Ukkonen. Approximate string matching with $q$-grams and maximal matches. *Theoretical Computer Science*, 1:191–211, 1992.

[Ukk93] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.

[Ukk95] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, Sep 1995.

[Ull77] J. Ullman. A binary $n$-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *The Computer Journal*, 10:141–147, 1977.

[UW93] E. Ukkonen and D. Wood. Approximate string matching with suffix automata. *Algorithmica*, 10:353–364, 1993. Preliminary version in Report A-1990-4, Dept. of Computer Science, Univ. of Helsinki, April 1990.

[Vid86] E. Vidal. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

[Vin68] T. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4:52–58, 1968.

[Wat95] M. Waterman. *Introduction to Computational Biology*. Chapman and Hall, 1995.

[Wat96] B. Watson. The performance of single and multiple keyword pattern matching algorithms. In *Proc. WSP'96*, pages 280–294. Carleton University Press, 1996.

[WB74] R. Wagner and T. Brown. Order-$n$ swap-extended correction of regular languages. Technical report, Systems & Information Science Program, Vanderbilt University, 1974.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proc. IEEE Symp. on Switching and Automata Theory*, pages 1–11. IEEE Press, 1973.

[WF74] R. Wagner and M. Fisher. The string to string correction problem. *J. of the ACM*, 21:168–178, 1974.

[WL83] W. Wilbur and D. Lipman. Rapid similarity searches in nucleic acid and protein data banks. In *Proc. of the National Academy of Sciences of the USA*, volume 80, pages 726–730, 1983.

[WM92a] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.

[WM92b] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, Berkeley, CA, USA, Winter 1992. USENIX Association.

[WMB94] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Van Nostrand Reinhold, New York, 1994.

[WMM95] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate regular expression matching. *J. of Algorithms*, 19(3):346–360, 1995. Submitted in 1992.

[WMM96] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996. Preliminary version as Technical Report TR29-36, Computer Science Dept., Univ. of Arizona, 1992.

[Wri94] A. Wright. Approximate string matching using within-word parallelism. *Software Practice and Experience*, 24(4):337–362, April 1994.

[WS78] R. Wagner and J. Seiferas. Correcting counter-automaton-recognizable languages. *SIAM J. on Computing*, 7:357–375, 1978.

[Yao79] A. Yao. The complexity of pattern matching for a random string. *SIAM J. on Computing*, 8:368–387, 1979.

[Yia93] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. ACM-SIAM SODA'93*, pages 311–321. SIAM Press, 1993.

[Zip49] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.