UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

Efficient Text Searching

Ricardo A. Baeza-Yates

CS-89-17

May 1989

Efficient Text Searching

by

Ricardo A. Baeza-Yates

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, 1989

Abstract

This thesis presents and analyses new algorithms for text searching. We place special emphasis on the average case of each algorithm, because we are interested in practical applications. We consider plain text and preprocessed text.

In preprocessed (indexed) text our main result is to show that automaton searching over a digital tree requires time sublinear in the size of the text for any regular expression; this is the first algorithm with a searching time complexity better than linear. We also show that for a restricted set of regular expressions, searching requires logarithmic time in the expected case. In all these algorithms, the time is independent of the size of the answer.

For searching in plain text, our main results are the following. First, we analyze the expected case of the best known string matching algorithms, obtaining bounds for the Knuth-Morris-Pratt algorithm, and the Boyer-Moore algorithm and several of its variations. We also present improvements to Boyer-Moore type algorithms, in particular for searching English text. Second, we analyze three new, simple, and fast algorithms to solve the problem of string matching with mismatches. These algorithms are eminently practical and can be extended to other variations of this problem. Finally, we present a new approach to pattern matching, based on a numerical representation of the state of the search. We apply this method to patterns including classes of symbols, complement, "don't care" symbols, multiple patterns, and mismatches. The search time is linear for most practical cases. One of the advantages of these algorithms is that they are suitable for hardware implementation.

Acknowledgements

There are several people who have made possible the successful completion of this work to whom I am indebted. First amongst these is my wife Susana, who decided to follow me in this challenge, and supported me with love and patience throughout all this time. Gracias, mi amor.

To my mother, Patricia Yates, for her constant love and support, and to all my family.

I have been fortunate in having Professor Gaston Gonnet as my thesis supervisor, and I would like to express my sincere gratitude for his valuable guidance, encouragement and friendship. I have benefited by learning from his experience.

To the other members of my thesis committee, Professors Alberto Apostolico, Frank Tompa, Derick Wood and Dan Younger, for their helpful criticism and careful reading. Special thanks to Per-Åke Larson, for his unconditional support, encouragement and friendship.

To all my friends during my graduate years at Waterloo for the useful discussions and all the fun provided by them. Special mention to Ricardo Dahab, Tony Gahlinger and Greg Rawlins.

Finally, the financial support from the Institute for Computer Research (ICR), the province of Ontario (through an Ontario graduate scholarship), the UW Centre for the New Oxford English Dictionary, the Information Technology and Research Centre (ITRC), the Department of Computer Science at the University of Waterloo, and the University of Chile are gratefully acknowledged.

This thesis was prepared using the LaTeX [Lam86] macros for TeX [Knu84], Pic [Ker82] for pictures, and GTS [LM85] for graphs. Many of the results presented in the thesis were checked using the Maple symbolic algebra system [CGG+88].

To Susana, Gonzalo, and Ignacio

"The Pacific Ocean fell off the
map. There was no place left.
It was so big, untidy and blue
that it did not fit anywhere.
That is why it was left
before my window."
PABLO NERUDA

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1878 H. PHILLIPS tr. *Poems fr. Spanish & German.* 19
And Baeza's tocsin note Bellows forth from brazen throat.
OED, **tocsin**

1.1 Motivation

Pattern matching and text searching are very important components of many problems, including text editing, data retrieval and symbol manipulation. Formally the problem can be defined as follows: Given a *text* string t and a *query* (pattern) q, locate either one occurrence (for example, the first), or all occurrences, of q in t; or return "none present".

We are interested in solving this problem for patterns expressed in various query languages, ranging from a simple string to regular expressions; and for either *plain text*, that is, just a piece of text; or *preprocessed text*, that is, a piece of text plus some kind of index. Our main motivation has been the work done as part of the New Oxford English Dictionary project at the University of Waterloo. One of the main problems of this dictionary is its size. For this reason we are mainly interested in efficient algorithms for the *average* case.

1.2 Previous Work

For plain text, there are efficient algorithms that solve particular cases in time linear in the size of the text in the worst case: Boyer-Moore [BM77] and Knuth-Morris-

Pratt [KMP77] for searching one string, Aho and Corasick [AC75] and Commentz-Walter [CW79] for searching a set of strings, Fischer and Paterson [FP74] for searching strings with "don't care" symbols, Pinter [Pin85] for searching strings with "don't care" and "complement" symbols, and Abrahamson [Abr87] for searching strings with classes of symbols.

If the text can be preprocessed, there are data structures that can be used as indexes to provide quick searching. For example, with a Patricia tree [Mor68], it is possible to search for all the occurrences of text sharing a common prefix in linear worst case time in the size of the prefix [Knu73, Gon83].

The question of whether it is possible to search for text matching a regular expression in time dependent only in the query size, and not in the text size, is still open [Gal85].

Our main goal is to find a suitable preprocessing that will allow us to build an index using at most $O(n)$ storage, where n is the size of the text and through which we can answer any query in average time $O(Pol(|query|) + \log n)$, that is, in time proportional to a polynomial in the size of the query plus the logarithm of the size of the text, but independent of the size of the answer. We achieve this goal for a large subclass of regular expressions.

By introducing automata searching over a Patricia tree, we obtain sublinear, but superlogarithmic search time in the size of the text for any regular expression.

1.3 The New Oxford English Dictionary

The *Oxford English Dictionary (OED)* is the largest and most comprehensive dictionary of the English language. Work for the OED started in 1857, and its first publication spanned from 1884 to 1928, through twelve volumes. A supplement to the dictionary was published in 1933. This supplement was replaced later by four volumes published between 1972 and 1986. The second edition of the OED was published in March of 1989.

In 1984, the Oxford University Press made public its decision to computerize the dictionary, with the participation of the University of Waterloo. After one year, a formal agreement was signed, and the Waterloo component of the New Oxford English Dictionary (New OED) project was started. The main goal of the project is to build a database system to maintain and access the dictionary data [BGT88].

The Dictionary has been structured by tagging the text with descriptive information. These tags include information about the structure itself, such as the scope of entries and quotations, and related attributes, for example to indicate font type. For a detailed description of the structure, we refer the reader to Kazman [Kaz86].

The OED poses one main problem to maintaining and searching it: its size. The second edition of the OED (1989) occupies almost 540Mb of secondary storage, defining approximately 616,500 words and terms. The printed version needs 20 volumes with 21,728 pages.

Given the size of the Dictionary, extracting information by linear search is only conceivable in batch mode. For interactive use, faster searching algorithms are needed. The solution taken by the project was an efficient implementation of Patricia trees [Mor68, Knu73, Gon83] that led to PAT, which provides fast string searching, longest repetitions searching, range searching, proximity searching, boolean operators, and more [Gon87, Faw89].

Another problem with the OED is how to extract information from it. Textual information is very different from common data applications, and conventional databases methods do not help in this case.

For example, a simple query may be: What is the meaning of "sesquipedalian"? This can be answered easily just by looking at that entry. But consider questions such as:

- How many words did Shakespeare introduce into the language between 1610-11?
- Is there a term to describe the battle between mice and frogs? Which works use this term?
- How many words of Spanish origin are there in the language?
- How do you name the action of writing alternatively from left to right and from right to left?

These and other similar questions [BCJ+85] cannot be answered without searching the whole dictionary. (In case the reader is interested, the answers to the above questions are 113, batrachomyomachy, at least 2322, and baustrophedon, respectively.)

One of the main motivations of the first part of this thesis is to provide efficient algorithms that, using a PAT index, allows users to search for more complex queries incorporating regular expressions. At the interactive level, PAT, by using regular expressions as a query language can answer some of the above questions. Hence, our work is a first step to answer complex queries efficiently in very large text databases.

By using PAT as a first approximation, and then using more specific algorithms, we can answer more complicated queries. This is one of the motivations for the second part of this thesis, where we design efficient and practical algorithms for plain text, for example, string searching with mismatches and/or "don't care" symbols.

To the best of our knowledge, all the results of this thesis for which there is no explicit reference are new. The most significant new results are presented in Chapters 3 and 7.

Chapter 2

Basic Concepts

1597 SHAKS. *Lover's Compl.* 2
From off a hill whose concaue wombe reworded
A plaintfull story from a sistring vale.
OED2, **reword, sistering**

1906 *Dialect Notes* III. 122
Quittin time, regular expression for the time to cease work.
OED2, **quitting-time**

In this chapter we present the basic concepts that we need throughout this thesis. We begin with a brief review of strings, languages, regular expressions, finite automata, text, and digital trees.

2.1 Strings

We use Σ to denote the *alphabet* (a set of symbols). We say that the alphabet is *finite* if there exist a bound in the size of the alphabet, denoted by $|\Sigma|$. Otherwise, if we do not know *a priori* a bound in the alphabet size, we say that the alphabet is *arbitrary* . A *string* over an alphabet Σ is a finite length sequence of symbols from Σ. The *empty string* (ϵ) is the string with no symbols. If x and y are strings, xy denotes the *concatenation* of x and y. If $w = xyz$ is a string, then x is a *prefix*, and z a *suffix* of w. The *length* of a string x ($|x|$) is the number of symbols in x. Any contiguous sequence of letters y from a string w is called a *substring*.

2.2 Regular Expressions

We use the usual definition of regular expressions (RE for short) defined by the operations of concatenation, union (+) and star or Kleene closure (*) [HU79]. A *language* over an alphabet Σ is a set of strings over Σ. Let L_1 and L_2 be two languages. The language $\{xy|x \in L_1 \textit{ and } y \in L_2\}$ is called the *concatenation* of L_1 and L_2 and is denoted by L_1L_2. If L is a language, we define $L^0 = \{\epsilon\}$ and $L^i = LL^{i-1}$ for $i \geq 1$. The *star or Kleene closure* of L, L^*, is the language $\bigcup_{i=0}^{\infty} L^i$. The *plus or positive closure* is defined by $L^+ = LL^*$.

We use $L(r)$ to represent the *set of strings* in the language denoted by the regular expression r. The *regular expressions* over Σ and the languages that they denote (*regular sets* or *regular languages*) are defined recursively as follows [HU79]:

- $\emptyset$ is a regular expression and denotes the empty set.
- ϵ (empty string) is a regular expression and denotes the set $\{\epsilon\}$.
- For each symbol a in Σ, a is a regular expression and denotes the set $\{a\}$.
- If p and q are regular expressions, then $p + q$ (union), pq (concatenation), and p^* (star) are regular expressions that denote $L(p) \cup L(q)$, $L(p)L(q)$, and $L(p)^*$, respectively.

To avoid unnecessary parentheses we adopt the convention that the star operator has the highest precedence, then concatenation, then union. All operators are left associative.

We also use:

- Σ to denote any symbol from Σ (when the ambiguity is clearly resolvable by context).
- $r?$ to denote zero or one occurrence of r (that is, $r? = \epsilon + r$).
- $[a_1..a_m]$ to denote a *range* of symbols from Σ. For this we need an *order* in Σ.
- $S_1 - S_2$ to denote *set difference*, that is, the set of symbols in S_1 that are not in S_2.
- $\overline{a}$ to denote any symbol in Σ except a (complement, that is, $\Sigma - \{a\}$).
- $r^{\leq k}$ to denote $\sum_{i=0}^{k} r^i$ (finite closure).

Examples: All the examples given here arise from the OED:

1. All citations to an author with prefix Scot followed by at most 80 arbitrary characters then by works beginning with the prefix Kenilw or Discov:

   $$\texttt{<A>Scot } \Sigma^{\leq 80} \texttt{ <W>(Kenilw + Discov)}$$

   where <> are characters in the OED text that denote tags (A for author, W for work).

2. All "bl" tags containing a single word consisting of lowercase alphabetical only:

   $$\texttt{<bl>[a..z]*</bl>}$$

3. All first citations accredited to Shakespeare between 1610-11:

   $$\texttt{<EQ>(<LQ>)?<Q><D>161(0+1)</D> <A>Shak}$$

4. All references to author W. Scott:

   $$\texttt{<A>((Sir}\flat\texttt{)? W)?}\flat\texttt{Scott }\flat\texttt{?</A>}$$

   where $\flat$ denotes a literal space.

We use regular languages as our query domain, and regular languages can be represented by regular expressions. Sometimes, we restrict the query to a subset of regular languages. For example, when searching in plain text, we have the exact string matching problem, where we only allow single strings as valid queries.

2.2.1 Formal Problem

Given a string t (the text), a regular expression q (the query), and information (optionally) obtained by preprocessing the pattern and/or the text, the problem consists of finding whether $t \in \Sigma^* q \Sigma^*$ (q for short) and obtaining some or all of the following information:

1. The location where an occurrence (or specifically the first, the longest, etc.) of q exists. Formally, if $t \in \Sigma^* q \Sigma^*$, find a position $m \geq 0$ such that $t \in \Sigma^m q \Sigma^*$. For example, the first occurrence is defined as the least m that fulfills this condition.
2. The number of occurrences of the pattern in the text. Formally, the number of all possible values of m in the previous category.
3. All the locations where the pattern occurs (the set of all possible values of m).

In general the complexities of these problems are different.

We assume that ϵ is not a member of $L(q)$. If it is, the answer is trivial.

2.3 Finite Automata

A finite automaton is a mathematical model of a system. The automaton can be in any one of a finite number of states and is driven from state to state by a sequence of discrete inputs. Figure 2.1 depicts an automaton reading its input from a tape.

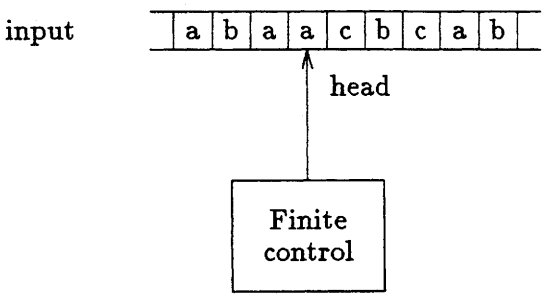

Figure 2.1: A finite automaton.

Formally, a *finite automaton* (FA) is defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ [HU79], where

- Q is a finite set of *states*,
- Σ is a finite *input alphabet*,
- $q_0 \in Q$ is the *initial* state,
- $F \subseteq Q$ is the set of *final* states, and
- δ is the (partial) *transition function* mapping $Q \times (\Sigma + \{\epsilon\})$ to zero or more elements of Q. That is, $\delta(q, a)$ describes the next state(s), for each state q and input symbol a; or is undefined.

A finite automaton starts in state q_0 reading the input symbols from a tape. In one move, the FA in state q and reading symbol a enters state(s) $\delta(q, a)$, and moves the reading head one position to the right. If $\delta(q, a) \in F$ we say that the FA has accepted the string written on its input tape up to the last symbol read. If $\delta(q, a)$ has an unique value for every q and a we say that the FA is *deterministic* (DFA); otherwise we say that it is *non-deterministic* (NFA).

The languages accepted by finite automata (either DFAs or NFAs) are regular languages. In other words, there exists a FA that accepts $L(r)$ for any regular expression r; and given a DFA or NFA, we can express the language that it recognizes as a RE. There is a simple algorithm [AHU74] that, given a regular expression r, constructs a NFA that accepts $L(r)$ in $O(|r|)$ time and space. There are also

algorithms to convert a NFA to a NFA without ϵ transitions ($O(|r|^2)$ states) and to a DFA ($O(2^{|r|})$ states in the worst case)[HU79].

A DFA is called *minimal* if has the minimum possible number of states. There exists an $O(|\Sigma| n \log n)$ algorithm to minimize a DFA with n states [HU79].

A finite automaton is called *partial* if the δ function is not defined for all possible symbols of Σ for each state. In that case, there is an implicit *error* state belonging to F for every undefined transition.

2.4 Text

2.4.1 Classes of Text

Text databases can be classified according to their update frequency as *static* or *dynamic*. Static text does not suffer changes, or its update frequency is very low. Thus, static databases may have new data added and can still be considered static. Examples of static text are historical data (no changes) and dictionaries.

For this type of database, it is worthwhile to *preprocess* the text and to *build indices* or other data structures to speed up the query time. The preprocessing time will be amortized during the time the text is searched before the data changes again. This is the case when searching the Oxford English Dictionary.

On the other hand, dynamic text is text that changes too frequently to justify preprocessing, for example, in text-editing applications. In this case, we must use search algorithms that scan the text sequentially, or that have efficient algorithms to update the index.

Text can be viewed as a very long string of data. Often text has little or no structure, and in many applications we wish to process the text without concern for the structure. Gonnet [Gon83] used the term *unstructured database* to refer to this type of data. Examples of such collections are: dictionaries, legal cases, articles on wire services, scientific papers, etc.

Text can instead be structured as a sequence of words. Each *word* is a string which does not include any symbol from a special separator set. For example, a "space" is usually an element of such a set.

We distinguish two kinds of text: random text and English text. In *random text* any string is the concatenation of symbols independently chosen from a finite alphabet Σ using a uniform distribution.

2.4.2 The Semi-Infinite String Model

Let's assume that the text to be searched is a single string and padded at its right end with an infinite number of null (or any special) characters. A *semi-infinite string* (*sistring*)[Gon88b] is the sequence of characters starting at any position of the text and continuing to the right. For example, if the text is

```
The traditional approach for searching a regular expression ...
```

the following are some of the possible sistrings:

```
The traditional approach for searching ...
he traditional approach for searching a ...
e traditional approach for searching a ...
onal approach for searching a regular ...
```

Two sistrings starting at different positions are always different. To guarantee that no one semi-infinite string be a prefix of another, it is enough to end the text with a unique end-of-text symbol that appears nowhere else [Knu73]. Thus, sistrings can be unambiguously identified by their starting position [AHU74, "position trees"]. The result of a lexicographic comparison between two sistrings is based on the text of the sistrings, not their positions.

2.5 Digital Trees

Efficient prefix searching can be done using indices. One of the best indices for prefix searching is a binary digital tree or binary trie constructed from the set of sistrings of the text.

Tries are recursive tree structures which use the digital decomposition of strings to represent a set of strings and to direct the searching. Tries were invented by de la Briandais [dlB59] and the name was suggested by Fredkin [Fre60], from information re*trie*val. If the alphabet is ordered, we have a lexicographically ordered tree. The root of the trie uses the first character, the children of the root use the second character, and so on. If the remaining subtrie contains only one string, that string's identity its stored in an external node.

Figure 2.2 shows a binary trie (binary alphabet) for the string "01100100010111..." after inserting the sistrings that start from positions 1 through 8. (In this case, the sistring's identity is represented by its starting position in the text.)

The *height* of a trie, is the number of nodes in the longest path from the root to an external node. The length of any path from the root to an external node is

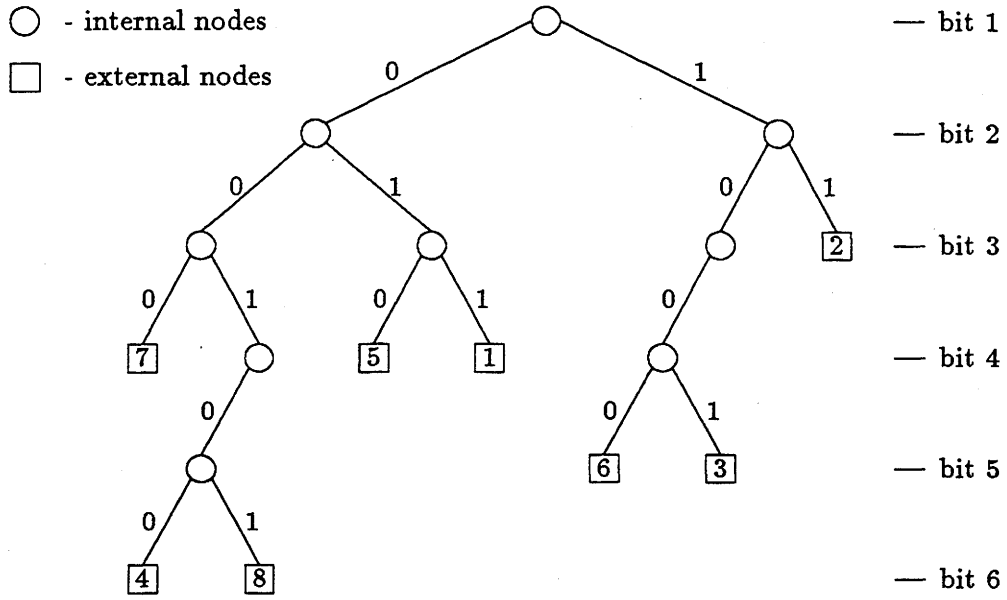

Figure 2.2: Binary trie (external node label indicates position in the text) for the first eight sistrings in "01100100010111...".

bounded by the height of the trie. On average the height of a trie is logarithmic for any square-integrable probability distribution [Dev82]. For a random uniform distribution, we have [Reg81]

$$\mathcal{H}(n) = 2\log_2(n) \; + \; o(\log_2(n))$$

for a binary trie containing n strings.

The average number of internal nodes inspected during a successful search in a binary trie with n strings is

$$\log_2 n \; + \; \frac{\gamma}{\ln 2} + \frac{1}{2} + P(\log_2 n) \; + \; O(1/n)$$

and for an unsuccessful search we have

$$\log_2 n \; + \; \frac{\gamma - 1}{\ln 2} + \frac{1}{2} + P(\log_2 n) \; + \; O(1/n)$$

The average number of internal nodes is

$$\frac{n}{\ln 2} + nP(\log_2 n) \; + \; O(1)$$

where $P(x)$ represents a periodic function with average value 0, period x, and very small absolute value [Knu73].

A *Patricia tree* [Mor68] is a trie with the additional constraint that single-descendant nodes are eliminated. This name is an acronym for "Practical Algorithm To Retrieve Information Coded In Alphanumerical". A counter is kept in each node to indicate which is the next bit to inspect. Figure 2.3 shows the Patricia tree corresponding to the binary trie in Figure 2.2.

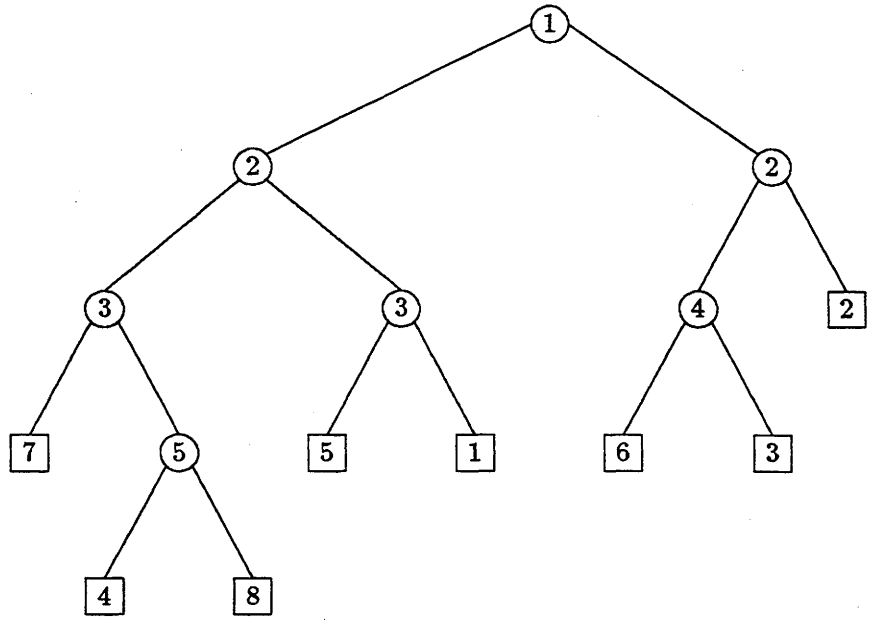

Figure 2.3: Patricia tree (internal node label indicates bit number).

For n sistrings, such an index has n external nodes (the n positions of the text) and $n-1$ internal nodes. Each internal node consist of a pair of pointers plus some counters. Thus, the space required is $O(n)$.

It is possible to build the index in $O(n\mathcal{H}(n))$ time, where $\mathcal{H}(n)$ denotes the height of the tree. As for tries, the expected height of a Patricia tree is logarithmic (and at most the height of the binary trie). We refer the reader to Gonnet [Gon84] for update algorithms on a Patricia tree.

A trie built using the sistrings of a string is also called *suffix tree*[AHU74]. Similarly, a Patricia tree is called a *compact suffix tree*. Although a random trie (*independent* strings) is different from a random suffix tree (*dependent* strings) in general, both models are equivalent when the symbols are uniformly distributed [AS87]. Moreover, the complexity should be the same for both cases, because $\mathcal{H}_{suffix\ tree}(n) \leq \mathcal{H}_{trie}(n)$ [AS87]. Simulation results suggest that the asymptotic ratio of both heights converges to 1 [AS89].

Chapter 3

Searching Using a Patricia Tree

1802-25 SYD. SMITH *Ess.* (ed. Beeton) 77
Here are 160 hours employed in the mere
digital process of turning over leaves.
OED, **digital**

1940 DYLAN THOMAS *Portrait of Artist as Young Dog* 58
He sat down in the road. 'I'm on a sledge,' he said,
'pull me, Patricia, pull me like an Eskimo.'
'Up you get, you moochin, or I'll take you home.'
OED2, **moochin**

In this chapter, we present algorithms for efficient searching of regular expressions on preprocessed text, using a Patricia tree as index. We obtain searching algorithms with logarithmic expected time in the size of the text for a wide subclass of regular expressions, and sublinear expected time for any regular expression. These are the first known algorithms to achieve these time complexities [BYG89a].

3.1 Introduction

The traditional approach for searching text for matches to a regular expression is to use the finite automaton that recognizes the language defined by the regular expression [Tho68] with the text as input. The main problems with this approach are:

- in general, all the text must be processed, giving an algorithm with running time linear in the size of the text [Riv77]. For many applications this is unacceptable.

- if the automaton is deterministic, both the construction time and number of states can be exponential in the size of the regular expression [HU79]. This is not a crucial problem in practice, because the size of the query is typically small and therefore can be treated as if bounded by a constant.

3.2 Prefix Searching

Efficient searching for all sistrings having a given prefix can be done using a Patricia tree. To search for a string (a word, a sequence of words, etc.) we start at the root of the tree following the searched string. Since in the search we may skip bits, one final comparison with the actual text is needed to verify the result. If the search ends inside the tree, the subtree rooted at the last inspected node contains the complete answer (see Figure 3.1). If the search string extends beyond the limits of the tree, then at most one position in the text matches. Thus, a search is done in $|string|$ steps. Similarly, if we look at the reversed text (that is, sistrings to the left), we can have *suffix* searching.

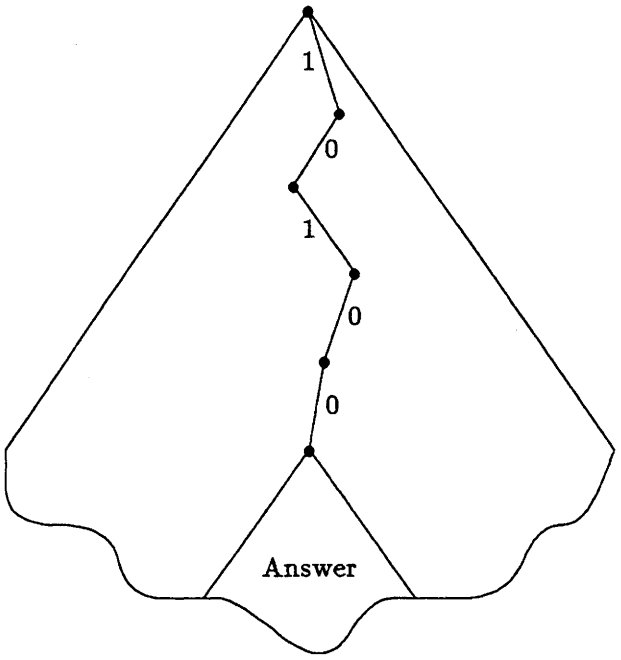

Figure 3.1: Prefix searching for "10100".

For many applications it is not necessary to store all the possible sistrings. For example, it may be sufficient to store sistrings that start at the beginning of a word.

This reduces the space by a factor proportional to the average word size, which for large databases may be a significant saving.

3.3 Range Searching and Longest Repetitions

Searching for a range of strings, and determining the longest repetition in the text starting with a given prefix also require logarithmic average time [Gon88b].

Range searching is to search for all strings in the text which compare lexicographically between two given prefixes. The search is done by finding the endpoint of each prefix, and then collecting all the subtrees between the endpoints. There are at most $2\mathcal{H}(n)$ subtrees in the answer. The subtrees can be collected while searching the endpoints.

The longest repetition in a text is the pair of sistrings that have the longest common prefix. The longest repetition of the whole text has length $\Omega(\mathcal{H}(n))$. By storing one bit per internal node, indicating which subtree is the taller, we can find the longest repetition for any subtree, hence for any common prefix.

3.4 Searching for a Set of Strings

We extend the form of the query to include searching for a finite set of strings. The simplest solution is to use prefix searching for every member of the set. However, we can improve the search time by recognizing common prefixes in our query.

Let the *complete prefix trie* be the trie of the set of strings, such that

- there are no truncated paths, that is, every word corresponds to a complete path in the trie; and
- if a word w is a prefix of another word x, then only w is stored in the trie. In other words, the search for w is sufficient to also find the occurrences of x.

Figure 3.2 shows an example.

Hence, the searching algorithm is

- Construct the complete prefix trie of the query using a binary alphabet.
- Traverse simultaneously the complete prefix trie and the Patricia tree of sistrings in the text (the "index"). All the subtrees in the index associated with terminal nodes in the complete prefix trie are the answer. As in prefix searching, because we may skip bits while traversing the index, a final comparison with the actual text is needed to verify the result.

The main property of a PRE is that there exists a unique finite set of words in the language, called *prewords*, such that:

- for any other string in the language, one of these words is a proper prefix of that string,
- the *prefix property* [HU79, page 121] hold; that is, no word in this set is a proper prefix of another word in the set.

Note that the size of the complete prefix trie of the prewords is linear in the size of the query. For example, the prewords of $ab(bc^* + d^+ + f(a + b))$ are *abb, abd, abfa,* and *abfb*. The complete prefix trie for this set is shown in Figure 3.2.

Table 3.1 gives descriptive recursive techniques to test if a query is a PRE and to find the prewords of a query. In both cases, the total time is linear in the size of the query. We use the notation $PRE(r)$ = true if r is a PRE, and $prewords(r)$ to denote the prewords of r.

$Query$	$PRE(Query)$	$prewords(Query)$
$\emptyset$	true	$\emptyset$
ϵ	true	$\{\epsilon\}$
$x \in \Sigma$	true	$\{x\}$
$r + s$	$PRE(r)$ and $PRE(s)$	$prewords(r) \bigcup prewords(s)$
$r\ s$	(r *is a string* and $PRE(s)$) or ($PRE(r)$ and ($\epsilon \in L(s)$))	$prewords(r)\ prewords(s)$
r^*	true	$\{\epsilon\}$

Table 3.1: Membership testing and prewords of a PRE-type query.

To search a PRE query, we use the algorithm to search for a set of strings, using the the prewords of the query. Because the number of nodes of the complete prefix trie of the prewords is $O(|query|)$, the search time is also $O(|query|)$. Therefore, we achieve our complexity goal (time independent of size of text and of answer set, space proportional to query size plus text size) for any PRE-type query.

3.6 Substring Analysis

Even though some queries can be represented by regular expressions, sometimes we can do much better than automata searching. For example, if the query is $a\ \ \Sigma^*\ \ b$

("a" followed by "b") we have two choices. One is to search for an a, and then follow every possible path in the index trying to find all possible b's. The second is to go to the text after each a, and sequentially search for b. In both cases, this is useless if there are no b's at all in the text. The aim of this section is to find from every query a set of necessary conditions that have to be satisfied. Hence, for many cases, this heuristic can save a lot of processing time later on. For this we introduce a graph representation of a query.

We define the *substring graph* of a regular expression r to be an acyclic directed graph such that each node is labelled by a string. The graph is defined recursively by the following rules:

- $G(\epsilon)$ is a single node labelled ϵ.
- $G(x)$ for any $x \in \Sigma$ is a single node labelled with x.
- $G(s+t)$ is the graph built from $G(s)$ and $G(t)$ with an ϵ-labelled node with edges to the source nodes and an ϵ-labelled node with edges from the sink nodes, as shown in Figure 3.3(a).
- $G(st)$ is the graph built from joining the sink node of $G(s)$ with the source node of $G(t)$ (Figure 3.3(b)), and re-labelling the node with the concatenation of the sink label and the source label.
- $G(r^+)$ are two copies of $G(r)$ with an edge from the sink node of one to the source node of the other, as shown in Figure 3.3(c).
- $G(r^*)$ is two ϵ-labelled nodes connected by an edge (Figure 3.3(d)).

Figure 3.4(a) shows the substring graph for the query $(ca^+s+h(a+e^*)m)(a+e)$. Each path in the graph represents one of the sequence of substrings that must be found for the pattern to match.

It is possible to construct this graph in time $O(|query|)$, using the above recursive definition.

Given a regular expression r, let $source(r)$ be the unique source node of $G(r)$ (no incoming edges). Similarly, let $sink(r)$ be the unique sink node of $G(r)$ (node without outgoing edges). All the node labels in a path between $source(r)$ and a node belonging to $sink(r)$ are substrings of a subset of words in $L(r)$. For example, in Figure 3.4(a) the strings ca, as and a are substrings of the subset ca^+sa of $L(r)$. Therefore, each label in $G(r)$ is a *necessary condition* for a non empty subset of words in $L(r)$. In other words, any word in the regular set described by r contains all the labels of some path in $G(r)$ as substrings.

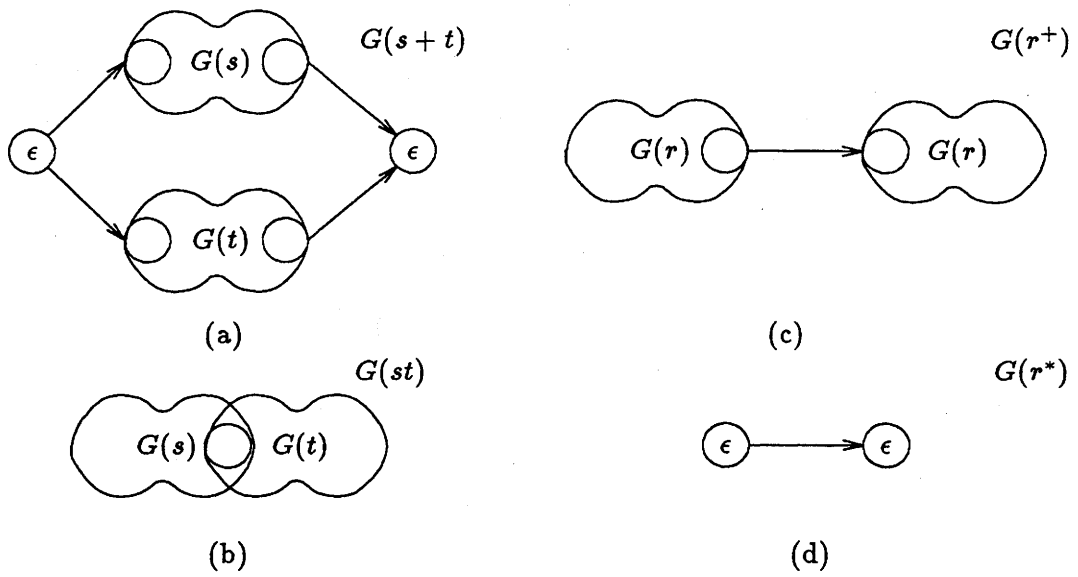

Figure 3.3: Recursive definition of a substring graph.

We explain the use of the substring graph through an example. Suppose that we have the query $q = (ab + ef + ij + op + uv)\ \Sigma^*\ (de + hi + no + tu)$. The substring graph $G(q)$ is shown in Figure 3.4 (b).

After building $G(q)$, we search for all node labels in $G(q)$ in our index of sistrings, determining whether or not that string exists in the text. This step requires $O(|q|)$ time, because the sum of the label lengths is $O(|q|)$. For example, the number of occurrences for each label in our example for a text sample is given beside each node in Figure 3.4(b).

For all non-existent labels, we remove:

- the corresponding node,
- adjacent edges, and
- any adjacent nodes (recursively) for which all incoming edges or all outgoing edges have been deleted.

In the example of Figure 3.4(b), we remove the node labelled by ij and the adjacent edges. This reduces the size of the query. Such reduction happens in practice when we have long labels or erroneously entered queries.

Second, from the number of occurrences for each label we can obtain an upper bound on the size of the final answer to the query. For adjacent nodes (serial, or

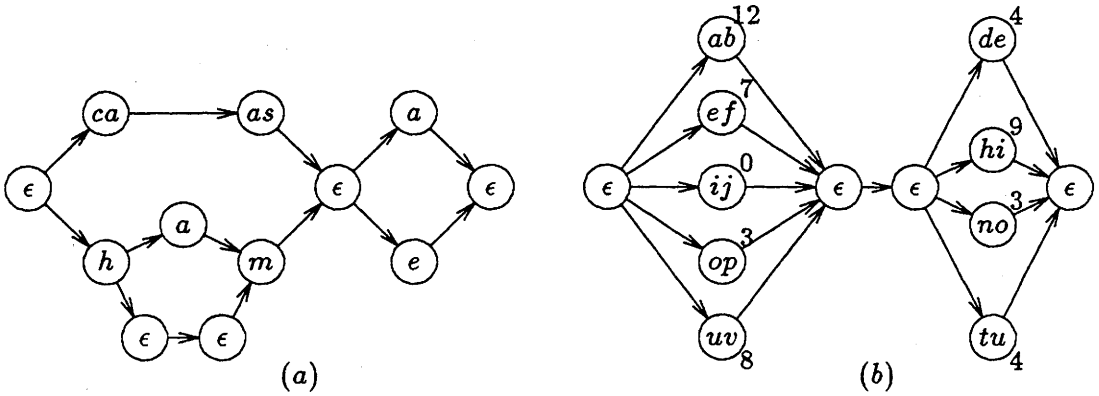

Figure 3.4: Substring graph of two queries.

and, nodes) we multiply both numbers, and for parallel nodes (*or* nodes) we add the number of occurrences.

At this stage, ϵ-nodes are simplified and treated in a special way:

- consecutive serial ϵ-nodes are replaced by a single ϵ-node (for example, the lowest nodes in Figure 3.4(a)),
- chains that are parallel to a single ϵ-node, are deleted (for example, the leftmost node labelled with a in Figure 3.4(a)), and
- the number of occurrences in the remaining ϵ-nodes is defined as 1 (after the simplifications, ϵ-nodes are always adjacent to non-ϵ-nodes, since ϵ was assumed not to be a member of the query).

In our example, the number of occurrences for Figure 3.4(b) can be bounded by 600. Similar to search strategies in inverted files [Knu73, Bat79], this bound is useful in deciding future searching strategies for the rest of the query.

3.7 Automaton Searching

In this section we present the only algorithm, to our knowledge, which can search for arbitrary regular expressions in time sublinear in n on the average. For this we simulate a DFA in a binary trie built from all the sistrings of a text.

The main steps of the algorithm are:

(a) Convert the regular expression passed as a query into a minimized DFA which may take exponential space/time with respect to the query size but is independent of the size of the text [HU79]. Next eliminate outgoing transitions

from final states (see justification in step (d)). This may induce further minimization.

(b) Convert character DFAs into binary DFAs using any suitable binary encoding of the input alphabet, each state will then have at most two outgoing transitions, one labelled 0 and one labelled 1.

(c) Simulate the binary DFA on the binary trie from all sistrings of text using the same binary encoding as in step (b). That is, associate the root of the tree with the initial state, and, for any internal node associated with state i, associate its left descendant with state j if $i \rightarrow j$ for a bit 0, and associate its right descendant with state k if $i \rightarrow k$ for a 1 (see Figure 3.5).

(d) For every node of the index associated with a final state, accept the whole subtree and halt the search in that subtree. (For this reason, we do not need outgoing transitions in final states).

(e) On reaching an external node, run the remainder of the automaton on the single string determined by this external node.

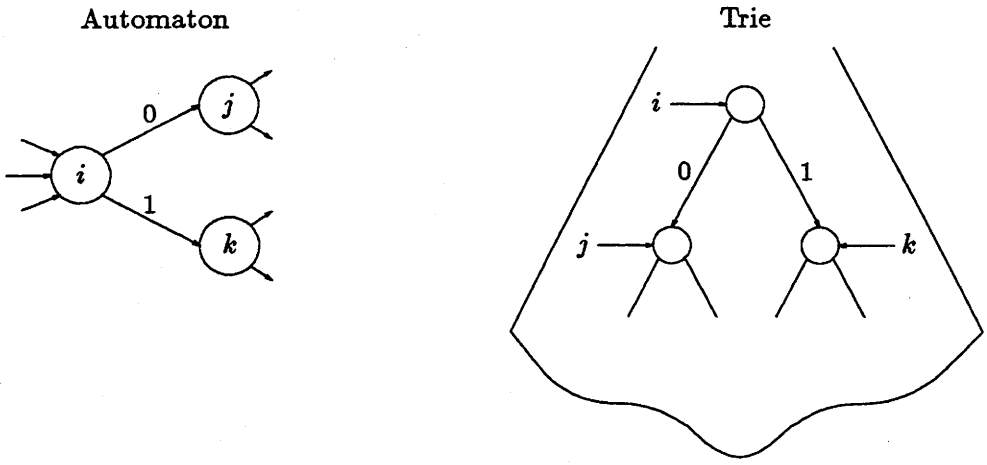

Figure 3.5: Simulating the automaton on a binary digital tree.

To adapt the algorithm to run on a Patricia tree, wherever we skip bits, it is necessary to read the corresponding text to determine the validity of each "skipped" transition: the sistring starting at any position in the current subtree may be used.

A depth-first traversal to associate automaton states with trie nodes ensures $O(\mathcal{H}(n))$ space for the simulation.

With some changes, the same algorithm is suitable for non-deterministic finite automata. In this event, for each node in the tree we have at most $O(|query|)$

active states during the simulation. Hence, we need $O(|query|\mathcal{H}(n))$ space for the simulation.

However we gain the advantage that the NFA has only $O(|query|)$ states and transitions [HU79]. Hence, the potential exponential size of the DFA is avoided.

3.7.1 Analysis

We analyze the algorithm for a random trie under the independent model. We already know that the independent model is equivalent to the dependent model for the random uniform case as pointed out in Section 2.5.

Let the initial state of the DFA be labelled 1. Let $\mathbf{H}$ be the incidence matrix of the DFA (that is, h_{ij} is the number of transitions from state i to state j) and $\vec{F}$ be a constant vector such that F_i is 1 for all i.

Lemma 3.7.1 *Let $\vec{N}(n)$ be the vector $[N_1(n), \ldots, N_s(n)]$, where $N_k(n)$ is the expected number of internal nodes visited when a DFA using state k as initial state is simulated on a binary trie of n random strings, and s is the number of states of the DFA. Then we have*

$$\vec{N}(n) = \sum_{k\geq 0} \tau_{n,k} \mathbf{H}^k \vec{F} \ ,$$

where

$$\tau_{n,k} = 1 - \left(1 - \frac{1}{2^k}\right)^n - \frac{n}{2^k}\left(1 - \frac{1}{2^k}\right)^{n-1} \ .$$

Proof: For a non-final state i, the expected number of internal nodes visited starting from it in a binary trie of n random strings can be expressed as $(n > 1)$

$$N_i(n) = 1 + \frac{1}{2^n} \sum_{k=0}^{n} \binom{n}{k} (N_j(k) + N_l(n-k)) \ ,$$

where $\delta(0, i) = j$ and $\delta(1, i) = l$. This is similar to the analysis of the expected number of nodes of a binary trie [Knu73]. For final states we have $N_f(n) = 1$ for $n > 1$. For undefined states (when $\delta(x, i)$ is not defined) we define $N_{undef}(n) = 0$. The initial conditions are $N_i(0) = N_i(1) = 0$ for any state i.

Introducing exponential generating functions in the above equation, that is,

$$N_i(z) = \sum_{n\geq 0} N_i(n) \frac{z^n}{n!} \ ,$$

we obtain

$$N_i(z) = e^{z/2}(N_j(z/2) + N_l(z/2)) \ + \ e^z - 1 - z \ .$$

Writing all the equations as a matrix functional equation, we have

$$\vec{N}(z) = e^{z/2}\mathbf{H}\vec{N}(z/2) \; + \; f(z)\vec{F} \; ,$$

where $f(z) = e^z - 1 - z$.

This functional equation may be solved formally by iteration [FP86], obtaining

$$\vec{N}(z) = \sum_{k\geq 0} e^{z(1-1/2^k)} f(z/2^k)\mathbf{H}^k\vec{F} \; .$$

From here, it is easy to obtain $\vec{N}(n) = n![z^n]\vec{N}(z)$ using the series expansion of e^x, where $[x^n]P(x)$ denotes the coefficient in x^n of the polynomial $P(x)$, from which the result follows. ■

From the previous lemma, we can obtain the exact number of nodes visited by the DFA for any fixed n. Note that the i-th element of $\mathbf{H}^k\vec{F}$, represents all possible paths in the DFA of length k that finish in state i. The next step is to obtain the asymptotic value of $N_1(n)$. This is based on the analysis of partial match queries in k-d-tries of Flajolet and Puech [FP86]. In fact, their analysis is similar to a DFA with a single cycle of length k.

Lemma 3.7.2 *The asymptotic value of the expected number of internal nodes visited is*

$$N_1(n) = \; \gamma_1(\log_2 n) \;\; (\log_2 n)^{m_1-1} n^{\log_2 |\lambda_1|} + O((\log_2 n)^{m_1-2} n^{\log_2 |\lambda_1|} + \log_2 n) \; ,$$

where $2 \geq |\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_t| \geq 0$, $\lambda_i \neq \lambda_j$ *are the eigenvalues of* $\mathbf{H}$ *with multiplicities* $m_1, \ldots, m_t$, s *is the order of* $\mathbf{H}$ *(number of states* $= \sum_i m_i = s$*), and* $\gamma_1(x)$ *is an oscillating function of* x *with period 1, constant mean value and small amplitude.*

Proof: Decomposing $\mathbf{H}$ in its upper normal Jordan form [Gan59], we have

$$\mathbf{H} = \mathbf{P} \; \mathbf{J} \; \mathbf{P}^{-1} \; ,$$

where $\mathbf{J}$ is a block diagonal matrix of the form

$$\mathbf{J} = \begin{bmatrix} J_1 & 0 & .. & .. \\ 0 & J_2 & 0 & .. \\ .. & .. & .. & .. \\ .. & .. & 0 & J_t \end{bmatrix} \; ,$$

J_i is a $m_i \times m_i$ square matrix of the form

$$J_i = \begin{bmatrix} \lambda_i & 1 & 0 & .. \\ 0 & \lambda_i & 1 & 0 \\ .. & .. & .. & 1 \\ .. & .. & 0 & \lambda_i \end{bmatrix} ,$$

and $\mathbf{P}$ has as columns the respective eigenvectors. Then

$$\mathbf{H}^k = \mathbf{P}\mathbf{J}^k\mathbf{P}^{-1} ,$$

where $\mathbf{J}^k$ is the block diagonal matrix $[J_i^k]$, and each J_i^k is of the form [Gan59]:

$$J_i^k = \begin{bmatrix} \lambda_i^k & k\lambda_i^{k-1} & \frac{k(k-1)}{2}\lambda_i^{k-2} & .. & \binom{k}{m_i-1}\lambda_i^{k+1-m_i} \\ 0 & \lambda_i^k & k\lambda_i^{k-1} & .. & \binom{k}{m_i-2}\lambda_i^{k+2-m_i} \\ 0 & 0 & \lambda_i^k & .. & \binom{k}{m_i-3}\lambda_i^{k+3-m_i} \\ .. & .. & .. & .. & .. \\ .. & .. & .. & 0 & \lambda_i^k \end{bmatrix} .$$

Therefore

$$\vec{N}(n) = \sum_{k\geq 0} \tau_{n,k}\mathbf{H}^k\vec{F} = \mathbf{P}\left(\sum_{k\geq 0} \tau_{n,k}\mathbf{J}^k\right)\mathbf{P}^{-1}\vec{F} .$$

Then, we have summations of the form

$$S_j = \sum_{k\geq 0} \tau_{n,k} \binom{k}{j} \lambda^{k-j} .$$

The convergence of S_j is guaranteed by the fact that, for fixed n and large k we have

$$\tau_{n,k} \approx 1 - e^{-n/2^k} - \frac{n}{2^k}e^{-(n-1)/2^k} = O\left(\frac{n^2}{4^k}\right) .$$

The asymptotic value of S_0 $(\lambda > 1)$ has already been obtained by Flajolet and Puech [FP86] as

$$S_0 = \gamma(\log_2 n)n^{\log_2 \lambda} + O(1) ,$$

where $\gamma(x)$ is a periodic function of x with period 1, mean value depending only on λ, and with a small amplitude. For details about the exact computation of the function γ using Mellin transform techniques, we refer the reader to Flajolet and Puech [FP86].

The asymptotic value of S_j $(\lambda > 1)$ is obtained in a similar way, namely

$$S_j = \frac{\gamma(\log_2 n)}{j!}\left(\frac{\log_2 n}{\lambda}\right)^j n^{\log_2 \lambda} + O(S_{j-1}) .$$

If $\lambda = 1$, then $S_j = O((\log_2 n)^{j+1})$. Then, for λ_i we have that the dominant term is

$$\gamma_i(\log_2 n)\ (\log_2 n)^{m_i-1}\, n^{\log_2 \lambda_i}\ ,$$

where we include the constant

$$\frac{1}{(m_i-1)!\lambda_i^{m_i-1}}$$

as part of the function γ_i. The highest order term of $N_1(n)$ is given by λ_1. In the case that there is more than one eigenvalue with the same modulus, λ_1 is the one with largest multiplicity. ■

Example 1: Let A be the DFA of the regular expression $(0(1+0))^*1(1(1+0))^*0$ (see Figure 3.6).

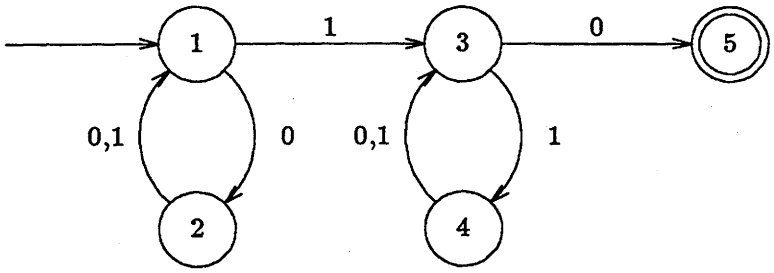

Figure 3.6: Deterministic finite automaton for $(0(1+0))^*1(1(1+0))^*0$.

The incidence matrix for A (state 1 is the initial state, and state 5 is the final state) is

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} .$$

The eigenvalues are $\sqrt{2}$ and $-\sqrt{2}$, each with multiplicity 2, and 0. The Jordan normal form for $\mathbf{H}$ is:

$$\mathbf{J} = \begin{bmatrix} \sqrt{2} & 1 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & -\sqrt{2} & 1 & 0 \\ 0 & 0 & 0 & -\sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} ,$$

The solution for this case is of the same order of the solution of the recurrence in Lemma 3.7.2. ■

The total time spent by the DFA is proportional to the number of nodes visited (internal and external) plus the number of comparisons performed to check the remainder of the single string corresponding to each external node. Thus, now we find the expected number of comparisons performed by the DFA when an external node is reached. As before, the DFA does not have outgoing transitions from final states.

Lemma 3.7.5 *The expected number of comparisons, $C(n)$, performed by a DFA while searching a random string of length m is bounded by a constant that is independent of m.*

Proof: The average number of comparisons in a random string of length ℓ starting in a non-final state i, can be expressed as

$$C_i(\ell) = 1 + \frac{1}{2}(C_j(\ell-1) + C_k(\ell-1)) \quad (\ell > 0)$$

where $\delta(0,i) = j$ and $\delta(1,i) = k$ [Knu73]. For final states we have $C_f(\ell) = 1$ for $\ell > 0$. For undefined states (when $\delta(x,i)$ is not defined) we define $C_{undef}(\ell) = 0$. The initial conditions are $C_i(0) = 0$ for any state i.

In a matrix form, we have

$$\vec{C}(\ell) = \frac{1}{2}\mathbf{H}\vec{C}(\ell-1) \; + \; \vec{F} \; .$$

Solving by iteration, we obtain

$$\vec{C}(\ell) = \sum_{j=0}^{\ell-1} \frac{1}{2^j}\mathbf{H}^j\vec{F} \; .$$

Using the same decomposition of Lemma 3.7.2 for $\mathbf{H}$, we have

$$\vec{C}(\ell) = \mathbf{P}\left(\sum_{j=0}^{\ell-1} \frac{1}{2^j}\mathbf{J}^j\right)\mathbf{P}^{-1}\vec{F} \; .$$

Therefore, we have summations of the form

$$S_j \; = \; \sum_{k\geq 0} \binom{k}{j} \frac{\lambda^{k-j}}{2^k} \; .$$

This geometric sum converges to a constant since $|\lambda| < 2$ (Lemma 3.7.3). For λ_1, we have that the constant is proportional to

$$\left(\frac{2}{2-\lambda_1}\right)^{m_1} ,$$

where m_1 is the multiplicity of λ_1. Moreover, because the norm of $\mathbf{H}$ is less than 2 (the maximum eigenvalue) [Gan59], we have

$$\vec{C}(\ell) \leq (\mathbf{I} - \frac{1}{2}\mathbf{H})^{-1}\vec{F} .$$

■

In Example 1, it is possible to bound $C_i(n)$ by 8 for all i. In the worst case, the number of comparisons could be exponential in the size of the query.

From the previous lemmas, we obtain our main theorem:

THEOREM **3.7.1** *The expected number of comparisons performed by a minimal DFA for a query q represented by its incidence matrix $\mathbf{H}$ while searching the trie of n random strings is sublinear, and given by*

$$O\left((\log_2 n)^t n^r\right) ,$$

where $r = \log_2 \lambda < 1$, $\lambda = \max_i(|\lambda_i|)$, $t = \max_i(m_i - 1,\ s.t.\ |\lambda_i| = \lambda)$, and the λ_is are the eigenvalues of $\mathbf{H}$ with multiplicities m_i.

Proof: The number of comparisons is proportional to the number of internal nodes plus the number of external nodes visited. In each external node, a constant number of comparisons is performed on average. ■

For NFAs, the same result holds multiplied by the length of the query because we have at most $O(|query|)$ states per active node. However, in this case the incidence matrix is computed using the ϵ-closure of each state.

Because the independent and dependent models are equivalent for random binary tries and random binary suffix trees [AS87], the complexity for a random suffix tree is the same. In practice we use a compact suffix tree. The asymptotic complexity of the algorithm in a compact suffix tree is also the same, because

- The asymptotic complexity of the algorithm in a complete balanced trie with the same number of sistrings is of the same order.
- For every suffix tree, the equivalent compact suffix tree has at most the same number of nodes.

3.8 Solving "Followed by"

A bad case for automata searching is when Σ^* is part of the query. This happens with the "followed by" search, that is, for example $s_1 \ \Sigma^* \ s_2$ which means an occurrence of s_1 followed by an occurrence of s_2.

The answer for these queries is given as a partially computed "join". Suppose that S_1 and S_2 are sets of positions where s_1 and s_2, respectively, occur. We define the solution as $S_1 \times_C S_2$ for any position in S_1 and S_2 such that the condition C is true. We ensure that every element of S_1 and S_2 is present in at least one solution pair. The condition C depends on s_1 and s_2. For example, if both are strings, C is $\{y \geq x + |s_1|, y \in S_2, x \in S_1\}$.

3.8.1 First Solution

We have two different solutions to this problem. The complexity of the first solution depends on the size of the answer.

In this solution, we augment the index by including for each internal node the *minimal* and *maximal* positions in the subtree rooted by that node.

Suppose s_1 and s_2 are strings. The algorithm is:

- Search for s_1 and s_2, obtaining solutions t_1 and t_2.
- If both solutions (subtrees) t_1 and t_2 are not empty, we traverse the subtree rooted by t_1 copying all positions such that $pos \leq \max_{pos}(t_2) - |s_1|$ to S_1.
- Do the same for t_2 using the condition $pos \geq \min_{pos}(t_1) + |s_1|$, obtaining S_2.
- The answer is S_1, S_2, and the condition $C = \{y \geq x + |s_1|, y \in S_2, x \in S_1\}$.

Because we already have $\min_{pos}(t_i)$ and $\max_{pos}(t_i)$ in each node, we can traverse both subtrees t_1 and t_2 only visiting $O(|S_1| + |S_2|)$ nodes.

For the general case, $r_1 \ \Sigma^* \ r_2$, for r_1 and r_2 regular expressions, the solution is:

- Search for s_1 and s_2, obtaining solutions t_1 and t_2. Now, t_1 and t_2 are, in general, a set of subtrees.
- If both solutions t_1 and t_2 are not empty, we traverse every subtree in t_1 copying all positions such that $pos \leq \max_{t \in t_2}(\max_{pos}(t) - height(root(t)))$ obtaining S_1. For each position in S_1 we must remember its length (depth in the tree).

- Do the same for t_2 using the condition $pos \geq \min_{t \in t_1}(\min_{pos}(t) + height(root(t)))$ obtaining S_2.
- The answer is S_1, S_2, and the condition $C = \{y \geq x + depth(x), y \in S_2, x \in S_1\}$.

The total search time is then $O(search(r_1) + search(r_2) + |S_1| + |S_2|)$. Note that the size of the complete solution may be as big as $|S_1||S_2|$, and hence $|S_1| + |S_2|$ is substantially better. On average, the size of $|S_i|$ is equal to $search(r_i)$; thus the total time is sublinear on average.

3.8.2 Second Solution

In our second solution, we trade storage for time. We obtain a solution which works in time independent of the size of the answer. For this, we include in each internal node all the positions (in order) that match with that node (leaves of the subtree rooted by that node). For each level of the tree, there are at most n positions. Hence, the space needed for this structure is $O(n\mathcal{H}(n))$, which is $O(n \log n)$ on average, rather than needing $O(n)$ space as in the previous solution.

Suppose we are searching for $s_1\ \Sigma^*\ s_2$, with s_1 and s_2 strings. The algorithm is

- Search for s_1 and s_2, obtaining the set of positions S_1 and S_2 respectively.
- Take the first position of S_1 (p) and search for $p + |s_1|$ in S_2, removing all the positions in S_2 less than $p + |s_1|$.
- Take the last position of S_2 (p) and search for $p - |s_1|$ in S_1, removing all the positions in S_1 greater than $p - |s_1|$.
- The answer is S_1, S_2, and the condition $C = \{y \geq x + |s_1|, y \in S_2, x \in S_1\}$.

Because the sets S_1 and S_2 are ordered, the last two steps of the algorithm require logarithmic time. Thus, the total number of internal nodes inspected on average is

$$\min(|s_1|, \mathcal{H}(n)) + \min(|s_2|, \mathcal{H}(n)) + \log_2 |S_1| + \log_2 |S_2| \ = \ O(\log n)$$

A similar approach works for $r_1\ \Sigma^*\ r_2$, for r_1 and r_2 regular expressions, considering all subtrees in the answer for r_1 and r_2. The total time is $O(search(r_1) \log S_1 + search(r_2) \log S_2)$ and $\log S_1 + \log S_2 = O(\log n)$. Hence, the average time is sublinear.

3.9 Merging the Algorithms

Our final searching algorithm merges all the results presented in this chapter:

- Detect the type of the query: string, PRE, "followed by", or RE.
- Perform the substring analysis, removing nonexistent subqueries.
- Search the query using the corresponding algorithm: string, PRE, Σ^* , or RE.

Figure 3.7 shows the searching process

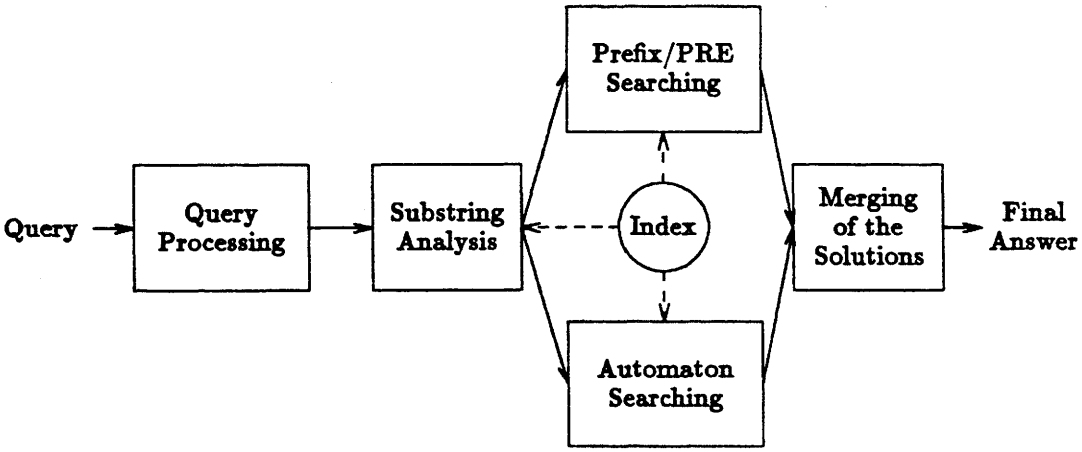

Figure 3.7: Query processing.

Chapter 4

Average Case Analysis of String Matching Algorithms

1851 in W. Pratt *Colonial Experiences* 234 (Morris)
[In the 'Dream of a Shagroon', which bore the date..April 1851,..
the term] 'pilgrim' [was first applied to the settlers].
OED2, **pilgrim**

In this chapter we study the average case of several string matching algorithms for plain text. We obtain analytical results for random text and derive an exact analysis for the naive algorithm and for different versions of the Boyer-Moore algorithm. Also, an approximate analysis is given for the Knuth-Morris-Pratt algorithm. Experimental results for the algorithms support the analyses and show that a random text model is a good approximation of English text[BY89c].

4.1 Introduction

The string matching problem consists of finding all occurrences (or the first occurrence) of a pattern in a text, where the pattern and the text are strings over some alphabet. In this chapter we are interested in reporting all the occurrences. It is well known that to search for a pattern of length m in a text of length n (where $n > m$) the search time is $O(n)$ in the worst case (for fixed m). Moreover, in the worst case at least $n - m + 1$ characters must be inspected [Riv77]. However, for different algorithms the constant in the linear term can be very different. For example, in the worst case, the constant multiple in the naive algorithm is m, whereas for the Knuth-Morris-Pratt [KMP77] algorithm it is two. On the other hand, the average

case of these algorithms has not been dealt with well. All previous analyses assume an approximate model or are not formally developed.

We derive an exact analysis for the naive algorithm, the Karp-Rabin [KR87] algorithm (a probabilistic one) and for different variants of the Boyer-Moore [BM77] algorithm in a random text model. An upper bound and an approximate analysis is derived for the Knuth-Morris-Pratt algorithm. Extensive experimental results support these analyses. Also, these results are compared against experimental results for English text. For patterns of length two, we provide exact analyses of almost all known algorithms. We also include some optimal algorithms for this case.

In this chapter and subsequent chapters, we use the C programming language [KR78] to present our algorithms.

4.2 Preliminaries

We use the following notation:

- n: Length of the text.
- m: Length of the pattern (string).
- c: Size of the alphabet.
- $\overline{C}_m$: Expected number of comparisons for a text of the same length as the pattern (m).
- $\overline{C}_n$: Expected number of comparisons for a text of length n.
- $\overline{I}_n$: Expected number of inspections for a text of length n. One character can be compared many times but inspected only once.

We are interested in the *average* number of comparisons between a character in the text and a character in the pattern (*text-pattern comparisons*) when finding *all* occurrences of the pattern in the text, where the average is taken uniformly with respect to strings of length n over a given alphabet.

Quoting Knuth *et al* [KMP77]: "It might be argued that the average case taken over random strings is of little interest, since a user rarely searches for a random string. However, this model is a reasonable approximation when we consider those pieces of text that do not contain the pattern, and the algorithm obviously must compare every character of the text in those places where the pattern does occur." Our experimental results show that this is the case.

4.2.1 Random Text

Let Σ be an alphabet of size c. Let p_i be the probability of choosing the i-th character from the alphabet. If we choose two characters independently from the alphabet, the probability that these characters are equal is

$$p_{equal} = \sum_{i=1}^{c} p_i^2 \ .$$

We define a *random string* of length ℓ as a string built as the concatenation of ℓ characters chosen independently and uniformly from Σ (that is, $p_i = \frac{1}{c}$, for all i). In the following we use random strings with a uniform distribution, however, all our results are valid for a non-uniform distribution if we use p_{equal} instead of $1/c$. In this sense, our random text model is similar to the one used in Knuth *et al* [KMP77] and Schaback [Sch88].

Let $text = t_1 t_2 \cdots t_n$ be a random string of length n and let $pattern = p_1 p_2 \cdots p_m$ be a random string of length m with $n \geq m$. In practice we have $n \gg m$.

Lemma 4.2.1 *In a comparison, the probability that two characters, one from the text and one from the pattern, are equal, is always $\frac{1}{c}$, unless we have other events conditioning the current event.*

Proof: This follows by definition, because in our model, both the text and the pattern are random strings over the same probability distribution. Therefore, each letter is chosen independently and, hence, $\sum_{i=1}^{c} p_i^2 = c \times 1/c^2 = 1/c$. ■

It is possible to extend the previous lemma to the case of a memoryless algorithm. In this case, the result also holds, because it does not matter if the character in the text, or the character in the pattern, or both, were compared before, because we are not recording (using) the result of the previous events.

Lemma 4.2.2 *The probability of finding a match between a random text of length m and a random pattern of length m is*

$$Prob\{match\} = \frac{1}{c^m} \ .$$

Proof: We have

$$Prob\{match\} = Prob\{t_1 = p_1 \wedge \cdots \wedge t_m = p_m\} = \prod_{i=1}^{m} Prob\{t_i = p_i\} = \frac{1}{c^m}$$

using the previous lemma. ■

Lemma 4.2.3 *The expected number of comparisons to decide if a random pattern of length m does or does not match a random text of length m is*

$$\overline{C}_m = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right) .$$

Proof: Because the strings are random strings, the order of the comparison of the m positions is irrelevant. Without loss of generality, suppose that the comparison order is from left to right. We have to perform one more comparison for each character matched, then

$$\overline{C}_m = 1 + \sum_{i=1}^{m-1} Prob\{t_1 = p_1 \wedge \cdots \wedge t_i = p_i\} ,$$

using Lemma 4.2.1, we have

$$\overline{C}_m = \sum_{i=0}^{m-1} \frac{1}{c^i} = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right) .$$

■

THEOREM **4.2.1** *The expected number of matches of a random pattern of length m in a random text of length n is*

$$E[matches] = \begin{cases} \frac{n-m+1}{c^m} , & \text{if } n \geq m. \\ 0, & \text{otherwise.} \end{cases}$$

Proof: In each m consecutive positions in the text, the probability of finding a match is given by $1/c^m$. Each one of these substrings is independent of the others by Lemma 4.2.1, and there are $n - m + 1$ substrings that may match the pattern. ■

In many cases, we need the expected value of the reciprocal of a random variable. Since in general $E\left(\frac{1}{x}\right) \neq \frac{1}{E(x)}$ we use the Kantorovich inequality [Cla82],

$$1 \leq E(x)E\left(\frac{1}{x}\right) \leq 1 + \frac{(x_{max} - x_{min})^2}{4x_{min}x_{max}} ,$$

where x_{min} and x_{max} denote the minimum and maximum values of the random variable x. If the variance of x converges to zero asymptotically in some parameter, then the lower bound is an equality (By the law of large numbers).

4.2.2 Markov Chains

We need some basic results from Markov chains. A stochastic process is a Markov process if the probability of one event depends only on the previous event. A Markov chain is a Markov process in discrete time with a discrete state space. In a Markov chain, each event is generally associated with a state. The above definition is equivalent to saying that the probability of a transition from time j to time $j+1$ depends on the state at time j[CM65].

Let $S = \{s_1, \cdots, s_r\}$ be the possible states in a Markov chain. Then, the *transition matrix* of the process is an $r \times r$ matrix defined as

$$\mathbf{P} = [p_{ij} = Prob\{i \rightarrow j | i\}] \ ;$$

that is p_{ij} is the conditional probability of transition from state i to state j given that the process is in state i. Let $\mathbf{p}^{(t)} = (p_1^{(t)}, \cdots, p_r^{(t)})$ be the state probability vector at time t (for example, $p_i^{(j)}$ is the probability of being in state i at time j). Then,

$$\mathbf{p}^{(j)} = \mathbf{p}^{(0)}\mathbf{P}^j \quad (j = 1, 2, \ldots) \ ,$$

where $\mathbf{p}^{(0)}$ is the initial vector of state probabilities.

We are interested in Markov chains with no absorbing states; that is, in all the states there exists at least one transition to another state. In this case, $\mathbf{p}^{(t)}$ converges to a stationary vector for large t. If $\boldsymbol{\pi}$ is the stationary vector of the state probabilities; that is

$$\lim_{j \rightarrow \infty} \mathbf{p}^{(j)} = \boldsymbol{\pi} \ ,$$

then $\boldsymbol{\pi}$ is the solution of the linear system of equations [CM65]

$$\boldsymbol{\pi}(\mathbf{P} - \mathbf{I}) = 0 \ , \ \sum_k \pi_k = 1 \ ,$$

where $\mathbf{I}$ is the identity matrix.

4.2.3 Experimental Results

The empirical data, for almost all algorithms, consists of results for two types of text: random text and English text. The two cost functions we measured were the number of comparisons performed between a character in the text and a character in the pattern, and the execution time. To determine the number of comparisons, 100 runs were performed. The execution time was measured while searching 1000 patterns. In each case, patterns of lengths 2 to 20 were considered.

In the case of random text, the text was of length 40000, and both the text and the pattern were chosen uniformly and randomly from an alphabet of size c. For c,

the values two (binary), four (DNA), 10, 30 (approximately the number of lowercase English letters) and 90 (approximately the number of printable ASCII characters) were considered.

For the case of English text we used a document of approximately 48000 characters, and the patterns were chosen at random from words inside the text, in such a way that a pattern was always a prefix of a word (typical searches). The alphabet used was the set of lower case letters, some digits, and punctuation symbols, giving 32 characters. Unsuccessful searches were not considered, because we expect unsuccessful searches to be faster than successful searches (fewer comparisons on average). The results for English text are not statistically significant, because only one text sample was considered. However, they show the correlation of searching patterns extracted from the same text, and we expect that other English text samples will give similar results.

Our experimental results agree with those presented by Davies [DB86] and Smit [Smi82].

4.3 Previous Results

The classic Knuth, Morris and Pratt algorithm, published in 1977 [KMP77], is the first algorithm for which the constant factor in the linear term, in the worst case, does not depend on the length of the pattern. It is based on preprocessing the pattern in time $O(m)$. In fact, the expected number of comparisons performed by this algorithm (search time only) is bounded by

$$n + O(1) \leq \overline{C}_n \leq 2n + O(1) .$$

In the same paper, it is shown that there exists an algorithm for which the expected worst case number of inspections is $O(\frac{\log_c m}{m} n)$ in random text, where c is the size of the alphabet. The authors conjectured that this was also a lower bound. For some patterns, the expected number of characters examined by this algorithm is $O(\frac{n}{m})$. Also, they presented an optimal algorithm for patterns of length two. The algorithm is optimal in the sense that it minimizes the average number of characters examined to find all occurrences of a pattern in the text. The expected number of inspections is

$$\overline{I}_n = \frac{c^2 + c - 1}{c(2c-1)} n \; + \; O(1) .$$

In the same year, the other classic algorithm was published by Boyer and Moore [BM77]. Their main idea is to search from right to left in the pattern. With this scheme, searching is faster on average. An analysis of the algorithm's average case behaviour is presented, based on some simplifying assumptions, however, no closed

form is provided. Experimental results support their analysis for alphabets of size greater than 30.

In 1979, Yao [Yao79] published the first paper on the average complexity of the string matching problem. He proved the conjecture raised in Knuth, Morris and Pratt's paper, showing that the average number of characters that need to be inspected in a random string of length n is

$$\overline{I}_n \geq \Theta\left(\frac{\lceil \log_c m \rceil}{m} n\right) ,$$

for almost all patterns of length m ($n > 2m$). This lower bound also holds for the "best case" complexity. A generalized basic algorithm with this average search time is presented. Note that $\overline{I}_n = 0$ in the limit when m or c tend to infinity.

Horspool in 1980 [Hor80] presented a simplification of the Boyer-Moore algorithm, and based on empirical results showed that this simpler version is as good as the original Boyer-Moore algorithm. Moreover, the same results show that this algorithm is better than algorithms which use a hardware instruction to find the occurrence of a designated character, for almost all pattern lengths. For example, the Horspool variation beats a variation of the naive algorithm (that uses a hardware instruction to scan for the character with lowest frequency in the pattern), for patterns of length greater than five.

The first (published) analysis of the naive algorithm appeared in 1984 by Barth [Bar84]. This work uses Markov chains and assumes that the probabilities of a transition from one state to another do not depend on the past. The expected number of comparisons needed to find the first match for the naive algorithm is

$$\overline{C}_{first\ match} = \frac{c^{m+1}}{c-1} - \frac{c}{c-1} ,$$

where c is the alphabet size. This paper, using a heuristic model, also derived an approximation for the Knuth-Morris-Pratt algorithm, given by

$$\overline{C}_{first\ match} \approx c^m + \frac{c^{m-1}}{c-1} + c - \frac{c}{c-1} .$$

From these results, Barth obtains an approximation for the ratio of the expected number of comparisons for the Knuth-Morris-Pratt and the naive algorithms, namely,

$$\frac{KMP}{naive} \approx 1 - \frac{1}{c} + \frac{1}{c^2}.$$

Later, two additional results appeared. The first one, by Schaback [Sch88], is an analysis of the average case of a variant of the Boyer-Moore algorithm under a

non-uniform alphabet distribution, obtained by approximating the algorithm by an automaton with an infinite number of states. His main result is that $\overline{C}_n/n < 1$ if

$$c(1 - \sum_{i=1}^{c} p_i^2) > 1 \ ,$$

where p_i is the probability of occurrence of the i-th symbol in the alphabet.

The second result, by Regnier [Reg88], is an average case analysis of the Knuth-Morris-Pratt algorithm. Using an algebraic approach to enumerate all possible patterns of length m by means of generating functions the following result is obtained:

$$\frac{\overline{C}_n}{n} \leq 1 + \frac{1}{c} - \frac{1}{c^m} \ ,$$

for large alphabets.

4.4 The Naive Algorithm

The naive, or brute force, algorithm is the simplest string matching method. The idea consists of trying to match any substring of length m in the text with the pattern (see Figure 4.1). Clearly this is a memoryless algorithm, because no information is recorded about how many characters are matched at each trial.

```
naivesearch( text, n, pat, m ) /* search pat[1..m] in text[1..n] */
char text[], pat[];
int n, m;
{
    int i, j, k, lim;

    lim = n-m+1;
    for( i = 1; i <= lim; i++ )     /* Search */
    {
        k = i;
        for( j=1; j<=m && text[k] == pat[j]; j++ ) k++;
        if( j > m ) Report_match_at_position( i-j+1 );
    }
}
```

Figure 4.1: The naive or brute force string matching algorithm.

THEOREM **4.4.1** *The expected number of text-pattern comparisons performed by the naive or brute-force algorithm when searching with a pattern of length m in a text of length n ($n \geq m$) is*

$$\overline{C}_n = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)(n - m + 1) \; + \; O(1) \; .$$

Proof: There are $n - m + 1$ substrings of length m in a text of length n, and in each one of them we perform $\overline{C}_m$ comparisons on average (Lemma 4.2.3). ■

Asymptotically, for m and n ($m \ll n$), we have

$$\frac{\overline{C}_n}{n} = \frac{c}{c-1} + O(c^{-m} + n^{-1})$$

This is drastically different from the worst case mn. Figure 4.2 shows the theoretical result, for some values of c, together with the empirical results. The agreement between both curves is very good.

4.5 The Knuth-Morris-Pratt Algorithm

The basic idea behind this algorithm is that each time a mismatch is detected, the "false start" consists of characters that we have already examined. We can take advantage of this information instead of repeating comparisons with the known characters [KMP77]. Moreover, it is always possible to arrange the algorithm so that the pointer in the text is never decremented. To accomplish this, the pattern is preprocessed to obtain a table that gives the next position in the pattern to be processed after a mismatch. The exact definition of this table (called *next* in Knuth *et al* [KMP77]) is

$$\begin{aligned} next[j] \;\; = \;\; & \max\{i | (pattern[k] = pattern[j - i + k] \text{ for } k = 1, \ldots, i - 1) \\ & \qquad \text{and } pattern[i] \neq pattern[j]\} \; , \end{aligned}$$

for $j = 1, \ldots, m$. In other words, we consider the maximal matching prefix of the pattern such that the next character in the pattern is *different* from the character of the pattern that caused the mismatch. This algorithm is presented in Figure 4.3. In the worst case, the number of comparisons is $2n + O(m)$. Further explanation of how to preprocess the pattern in time $O(m)$ to obtain this table can be found in the original paper [KMP77] or in [Sed83] (See Figure 4.4).

The basic idea of this algorithm was also discovered independently by Aho and Corasick [Aho80] to search for a set of patterns. However the space used and the preprocessing time to search for one string is improved in the Knuth-Morris-Pratt

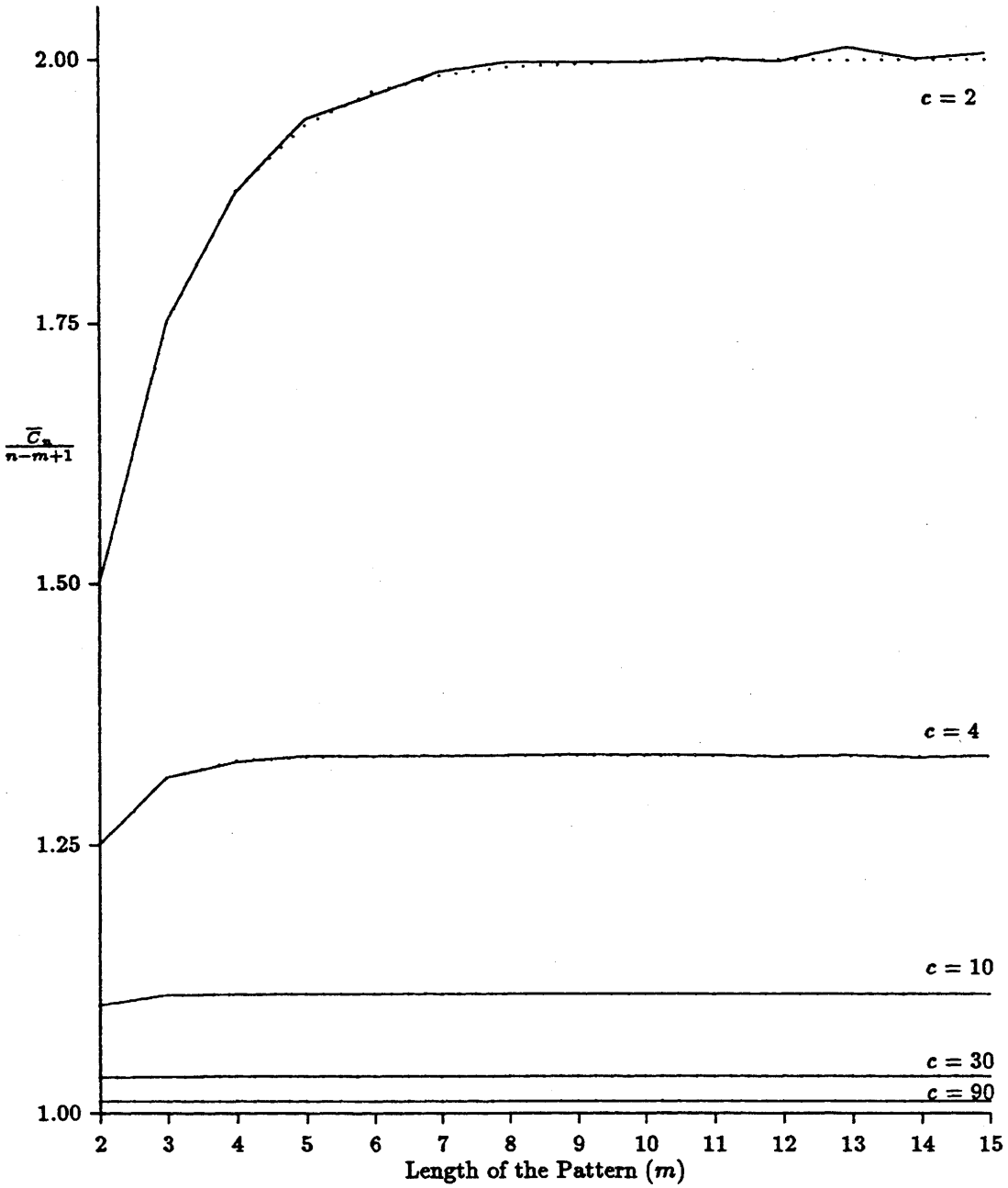

Figure 4.2: Simulation results for the naive algorithm in random text.
(The dotted lines represent the predicted results.)

```
kmpsearch( text, n, pat, m ) /* Search pat[1..m] in text[1..n] */
char text[], pat[];
int n, m;
{
    int j, k, resume, matches;
    int next[MAX_PATTERN_SIZE];

    pat[m+1] = CHARACTER_NOT_IN_THE_TEXT;
    initnext( pat, m+1, next );  /* Preprocess pattern */
    resume = next[m+1];
    next[m+1] = -1;
    j = k = 1;
    do {
        if( j==0 || text[k]==pat[j] )
        {
            k++; j++;
        }
        else j = next[j];
        if( j > m )
        {
            Report_match_at_position( k-j+1 );
            j = resume;
        }
    } while( k <= n );
pat[m+1] = END_OF_STRING;
}
```

Figure 4.3: The Knuth-Morris-Pratt algorithm.

algorithm. Variations that compute the *next* table on the fly are presented by Barth [Bar81] and Takaoka [Tak86].

The Knuth-Morris-Pratt algorithm is not a memoryless algorithm, in fact the entries in the *next* table act as different states during the search. Because the Knuth-Morris-Pratt algorithm is not memoryless, Barth's model [Bar84] is invalid. The Markov chains for the case $m = 2$ are shown in Figure 4.5, where p_3 is the stationary probability of visiting state 3. Each edge has two labels: the character compared and the probability of that event happening.

Using Markov chains we have obtained the following results.

Lemma 4.5.1 *The expected number of comparisons performed by the Knuth-*

```
initnext( pat, m, next )  /* Preprocess pattern of length m */
char pat[];
int m, next[];
{
    int i, j;

    i = 1; j = next[1] = 0;
    do
    {
        if( j == 0 || pat[i] == pat[j] )
        {
            i++; j++;
            if( pat[i] != pat[j] ) next[i] = j;
            else                   next[i] = next[j];
        }
        else j = next[j];
    }
    while( i <= m ) ;
}
```

Figure 4.4: Pattern preprocessing in the Knuth-Morris-Pratt algorithm.

Morris-Pratt algorithm in a text of length n is

$$\frac{\overline{C}_n}{n} = 1 + \frac{(c-1)^2}{c^3} + O(1/n) ,$$

for a pattern of length 2, and

$$\frac{\overline{C}_n}{n} = 1 + \frac{c^3 - c^2 - c + 1}{c^4} + O(1/n) ,$$

for a pattern of length 3.

Proof: First, we compute the steady state probabilities for the Markov chain of each possible kind of pattern (two and four different *next* tables for $m = 2$ and $m = 3$, respectively). Second, we compute $\overline{C}_n$ for the Markov chain of each pattern, and weight it with the pattern's probability of occurrence. These contributions are then summed to give the final result. For example, for $m = 2$, we have $\overline{C}_n = n$ for the pattern aa, and $\overline{C}_n = n/(1-p_3)$ for the pattern ab, where $p_3 = (c-1)/(c^2+c-1)$. Therefore, ignoring $O(1/n)$ terms, we have

$$\frac{\overline{C}_n}{n} = \frac{1}{c} + \frac{c-1}{c(1-p_3)} = 1 + \frac{(c-1)^2}{c^3} ,$$

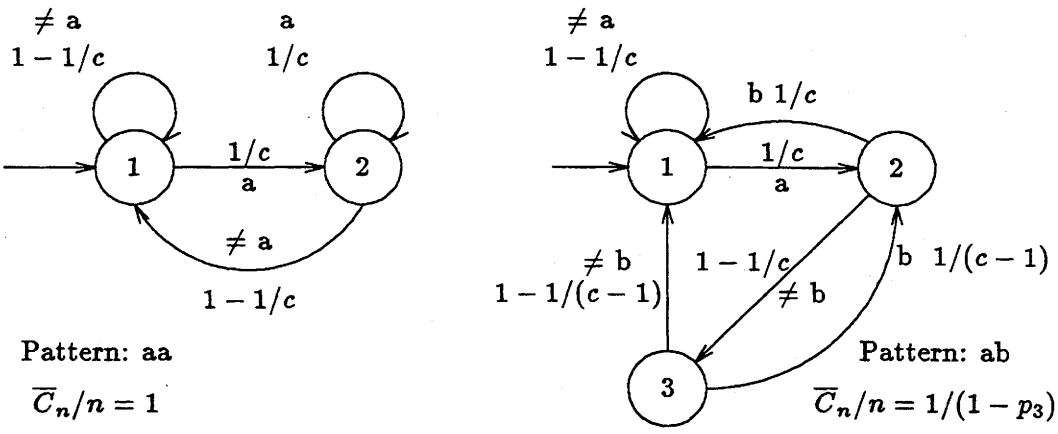

Figure 4.5: Markov chains for the Knuth-Morris-Pratt algorithm ($m = 2$).

because $p_{aa} = 1/c$ and $p_{ab} = 1 - 1/c$. ■

A somewhat surprising result is that for $m = 2$ and $m = 3$, the worst alphabet size is $c = 3$. Table 4.1 shows the empirical results contrasted with the exact theoretical values.

m	Result	$c = 2$	$c = 3$	$c = 4$	$c = 10$	$c = 30$	$c = 90$
2	Theoretical	1.125	1.148	1.1406	1.081	1.0312	1.01087
	Empirical	1.119	1.156	1.143	1.0740	1.0302	1.01092
3	Theoretical	1.1875	1.1975	1.1758	1.0891	1.0322	1.01099
	Empirical	1.171	1.188	1.163	1.0863	1.0313	1.01097

Table 4.1: Values of $\overline{C}_n/n$ for the Knuth-Morris-Pratt algorithm ($m \leq 3$).

For longer patterns, there are too many Markov chains. It is possible, however, to obtain a good upper bound by studying the structure of the Markov chains for the KMP algorithm. Remember that the Markov chains do not have absorbing states, thus, the probability of each state converges to a stationary value for large n (see Section 4.2.2). Note that a trivial lower bound for $\overline{C}_n$ is n.

THEOREM **4.5.1** *The expected number of comparisons of the KMP algorithm is bounded from above by*

$$\frac{\overline{C}_n}{n} \leq 2 - \frac{1}{c} + O(1/n) \ < 2 \ ,$$

for $c \leq \lceil \log_\phi m \rceil$, *and by*

$$\begin{aligned} \frac{\overline{C}_n}{n} &\leq 1 + \frac{(c-1)(c - \lceil \log_\phi m \rceil)}{c(c(c-1) + 1 - c\lceil \log_\phi m \rceil)} + O(1/n) \\ &= 1 + \frac{1}{c} + \frac{\lceil \log_\phi m \rceil - 1}{c^3} + O\left(\frac{\log^2 m}{c^4}\right) , \end{aligned}$$

for $c > \lceil \log_\phi m \rceil$, *with* $\phi = (1+\sqrt{5})/2 \approx 1.618$.

Proof: We reduce the Markov chain for any pattern to a 3-state chain. We consider the initial state and two meta-states: one being all the states where we match the pattern called the *skeleton* state, and the second being all the states outside the skeleton called the *outside* state, where we know more about the characters being compared. This Markov chain is depicted in Figure 4.6. The transitions of the reduced chain represent the relations between the reduced states. There is a transition from the skeleton state back to the initial state when $next[i] = 0$, for some $i > 1$. Similarly, there is a transition from the skeleton to itself when we match the last character and the pattern is periodic (when $next[m+1] > 1$). There is a transition from the outside state to the initial state when $next[i] = 0$, for some $i \geq 1$, and to the skeleton when we match a new character. There are no self transitions for the outside state, because there are no cycles in this set of states. Note that there is an extra comparison each time we are in the outside state, thus,

$$\frac{\overline{C}_n}{n} = \frac{1}{1 - p_{outside}} + O(1/n) .$$

Because we do not know the exact transitions, we have introduced three unknown probabilities: x, y, and z. We can bound these probabilities for any pattern. We have

$$0 \leq y \leq 1 - \frac{1}{c} ,$$

because in a state from the skeleton, the probability of not matching a character is $1 - 1/c$. Because there may be states in the skeleton that have a value of zero for $next$, this is an upper bound. Similarly, x is the probability of going back to the skeleton. This probability is zero for a non-periodic pattern and $1/c$ for a periodic one. Thus

$$0 \leq x \leq \frac{1}{c} .$$

For z, we have

$$\frac{1}{c-1} \leq z \leq 1 ,$$

since the probability of matching a character (and going back to the skeleton) is at least $1/(c-1)$, for every state outside the skeleton, because we know that the

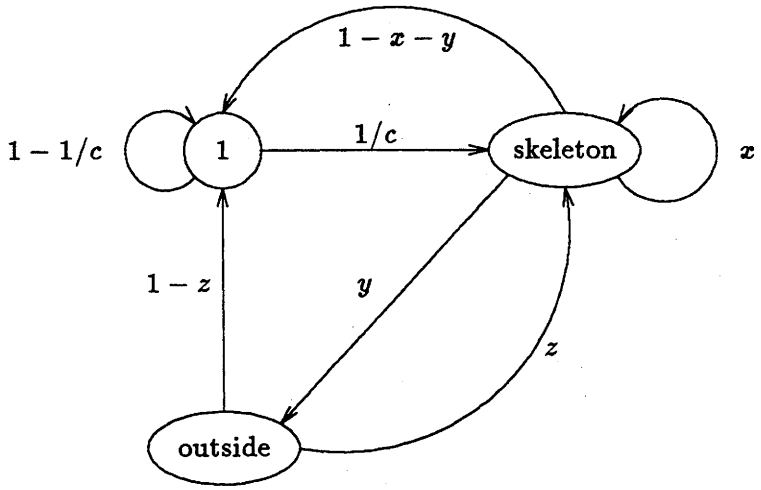

Figure 4.6: Reduced Markov chain for the Knuth-Morris-Pratt algorithm.

current character in the pattern is different from the character in the text. Solving this chain we get

$$p_{outside} = \frac{y}{c + 1 + y(1 - cz) - xc} .$$

This is maximized when $x = 1/c$, $y = 1 - 1/c$ and $z = 1$. Therefore, we have

$$\frac{1}{1 - p_{outside}} \leq 1 + \frac{c-1}{c} ,$$

proving the first part of the theorem. If $c > \lceil \log_\phi m \rceil$, then we know that the current character is different from at most $\lceil \log_\phi m \rceil$ characters. This is because at most $\log_\phi m$ comparisons are performed by the KMP algorithm before reading a new character [KMP77]. Therefore,

$$z \leq \frac{1}{c - \lceil \log_\phi m \rceil} .$$

Using this bound with $x = 1/c$ and $y = 1 - 1/c$ we obtain the second part of the theorem. ■

If the pattern is aperiodic, we can use $x = 0$ in the above proof. In this case, the expected number of comparisons is bounded above by

$$\frac{\overline{C}_n}{n} \leq 1 + \frac{c-1}{2c} + O(1/n) \; < \frac{3}{2} ,$$

for $c \leq \lceil \log_\phi m \rceil$, and by

$$\frac{\overline{C}_n}{n} \;\leq\; 1 + \frac{(c-1)(c - \lceil \log_\phi m \rceil)}{c(c^2 + 1 - (c+1)\lceil \log_\phi m \rceil)} \;+\; O(1/n)$$

$$= 1 + \frac{1}{c} - \frac{1}{c^2} + \frac{\lceil \log_\phi m \rceil - 1}{c^3} + O\left(\frac{\log^2 m}{c^4}\right) ,$$

for $c > \lceil \log_\phi m \rceil$.

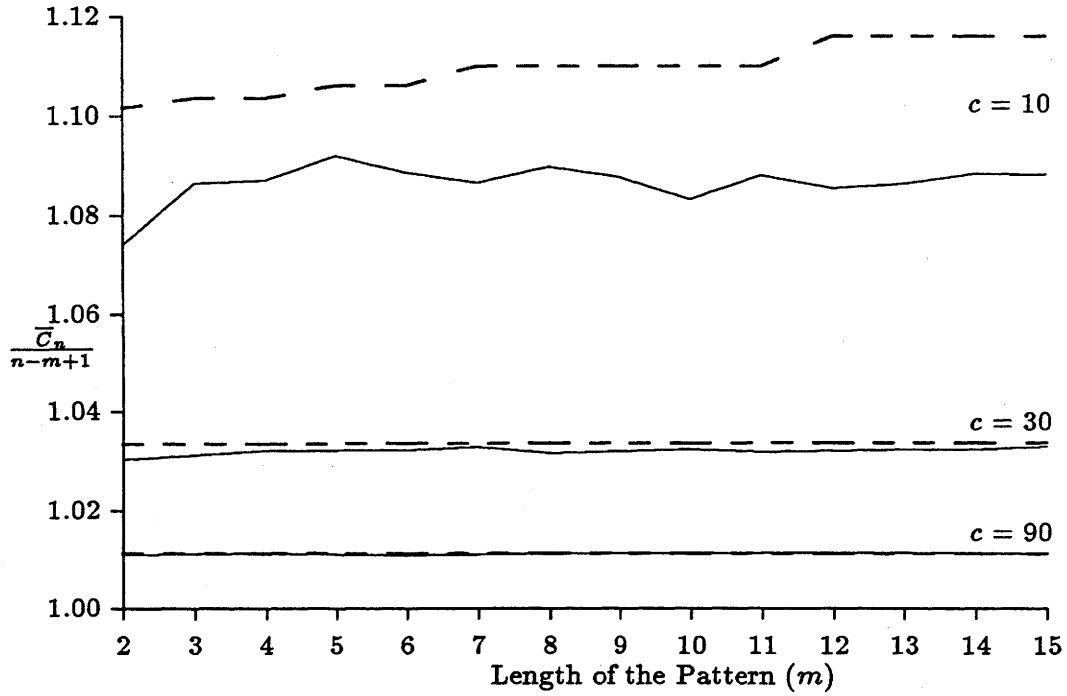

Figure 4.7: Theoretical upper bound (dashed lines) versus experimental results for the Knuth-Morris-Pratt algorithm.

In Figure 4.7 the upper bound is shown with the empirical results, for large alphabets. An exact analysis is difficult because of the many different patterns. One solution is to use only one Markov chain for all the patterns to model the algorithm. For example the Markov chain for the case $m = 2$ is shown in Figure 4.8. However, because the Markov chains are not *homogeneous* in the number of states; that is, not all the Markov chains for a fixed length pattern have the same number of states; the result from considering one Markov chain for all the patterns (*non-homogeneous* chain) is different from the result obtained by averaging the Markov chains for every possible pattern (as pointed out by Regnier [Reg88]). In the latter case, for $m = 2$ we have

$$\frac{\overline{C}_n}{n} = 1 + \frac{(c-1)^2}{c^3 + c - 1} + O(1/n) .$$

The difference in $\overline{C}_n/n$ is $O(c^{-3})$, for $m = 2$ and $m = 3$.

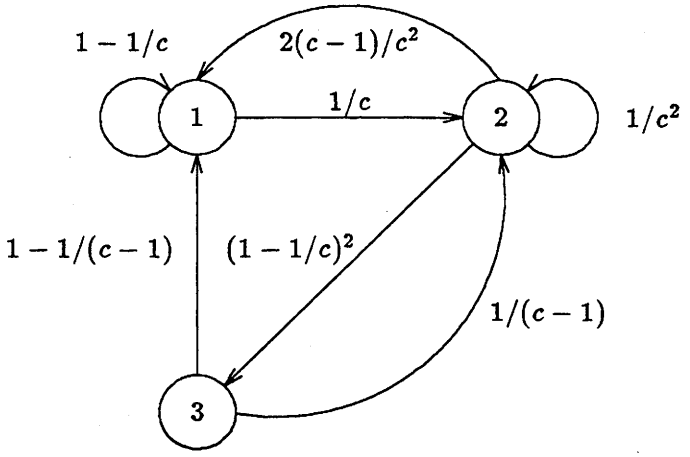

Figure 4.8: Non-homogeneous Knuth-Morris-Pratt Markov chain for $m = 2$.

In general the number of states is at most $m(m+1)/2$ for any pattern of length m. We derive an approximation with a linear number of states, which we believe is not far from the exact result for large alphabets.

The Markov chain has m states. In state j we have the following transitions:

- To state $j+1$, if $j < m$, with probability $1/c$ (match of a character) advancing the pointer in the text by one.
- To state i, if $j = m$ (after a match), with probability $Prob\{next[m+1] = i\}/c$, advancing the pointer in the text by one (if $i = 0$, go to state 1).
- To state i $(1 \leq i < j)$, if $1 < j < m$, with probability $(1-1/c)Prob\{next[j] = i\}$ without advancing the pointer in the text.
- To state 1, if $1 \leq j \leq m$, with probability $(1-1/c)Prob\{next[j] = 0\}$ advancing the pointer in the text by one.

The expected number of characters skipped per comparison (or *csc*, for short) is

$$\begin{aligned} E[csc] &= p_1 + \sum_{j=2}^{m}(1/c + (1-1/c)Prob\{next[j] = 0\})p_j \\ &= 1 - (1-1/c)\sum_{j=2}^{m}(1 - Prob\{next[j] = 0\})p_j \ . \end{aligned}$$

Suppose that we have a mismatch at position j of the pattern. If $next[j] \leq i$, for $i < j$, then $next[j] = i$ if the last $i-1$ characters of the partial match of length j are

a prefix of the pattern, and the j-th character in the pattern (the character that caused the mismatch) is different from the i-th character of the pattern. Hence,

$$Prob\{next[j] = i\} = \begin{cases} (1 - Prob\{next[j] \geq i+1\})(1-1/c)(1/c)^{i-1} & 1 \leq i < j \\ 0 & i \geq j \end{cases}$$

and

$$Prob\{next[j] = 0\} = 1 - \sum_{i>0} Prob\{next[j] = i\} .$$

Using the relation

$$Prob\{next[j] \geq i+1\} = \sum_{k=i+1}^{j-1} Prob\{next[j] = k\} ,$$

we obtain

$$Prob\{next[j] = i\} = \frac{c-1}{c^i}\alpha(j,i) ,$$

where

$$\alpha(j,i) = \begin{cases} 1 & i = j-1 \\ \prod_{k=i+1}^{j-1}\left(1 - \frac{c-1}{c^k}\right) & 0 < i < j-1 \end{cases}$$

and

$$Prob\{next[j] = 0\} = \frac{1}{c^{j(j+1)/2}} \prod_{k=2}^{j-1}(c^k - c + 1) .$$

The asymptotic probabilities are given by the set of equations

$$p_j = \frac{p_{j-1}}{c} + (1-1/c)\sum_{k=j+1}^{m} Pr\{next[k] = j\}p_k + Prob\{next[m+1] = j\}/cp_m ,$$

for $j > 1$ and $\sum_{j=1}^{m} p_j = 1$. The solutions for $m = 2$ and $m = 3$ are

$$\frac{\overline{C}_n}{n} = 1 + \frac{1}{c} - \frac{2}{c^2+1} ,$$

and

$$\frac{\overline{C}_n}{n} = 1 + \frac{(c-1)^2(c^4+c^3+1)}{c^2(c^5+c^3+c^2-c+1)} ,$$

respectively.

Solving the above system system of equations for large c, we obtain the following asymptotic approximation for the expected number of comparisons ($m > 2$)

$$\frac{\overline{C}_n}{n} \approx 1 + \frac{1}{c} - \frac{1}{c^2} - \frac{1}{c^3} + O(c^{-4}) ,$$

which is good for $c \geq 4$, as can be seen from the empirical results. Figure 4.9 shows the empirical results together with the approximation developed here. The difference between the approximation and our upper bound is $O(c^{-2})$, for $c > \lceil \log_\phi m \rceil$.

Barth's approximation of Knuth-Morris-Pratt algorithm is too optimistic as is shown in Figure 4.9, for the case $c = 4$. In fact, the average ratio between the two algorithms is better approximated by

$$\frac{KMP}{naive} \approx 1 - \frac{2}{c^2} .$$

This agrees with Regnier's result [Reg88], which implies for $m > 2$, that we have

$$\frac{KMP}{naive} \leq 1 - \frac{1}{c^2} .$$

4.6 The Boyer-Moore Algorithm

An improvement in the average search time is obtained by searching from the *right to the left* in the pattern [BM77]. The Boyer-Moore or BM algorithm positions the pattern over the leftmost characters in the text and attempts to match it from right to left. If no mismatch occurs, then the pattern has been found. Otherwise, the algorithm computes a shift; that is, an amount by which the pattern is moved to the right before a new matching attempt is undertaken.

The shift can be computed using two heuristics: the match heuristic and the occurrence heuristic. The *match* heuristic is obtained by noting that when the pattern is moved to the right, it has to

1. match *all* the characters previously matched, and
2. bring a *different* character to the position in the text that caused the mismatch.

The last condition is mentioned in Boyer-Moore's paper [BM77], but was introduced into the algorithm by Knuth *et al* [KMP77]. Following [KMP77] we call the original shift table dd, and the improved version $\widehat{dd}$. The formal definitions are

$$dd[j] = \min\{s+m-j | s \geq 1 \text{ and } ((s \geq i \text{ or } pattern[i-s] = pattern[i]) \text{ for } j < i \leq m)\} ,$$

for $j = 1, .., m$; and

$$\begin{aligned} \widehat{dd}[j] \;=\; & \min\{s + m - j | s \geq 1 \text{ and } (s \geq j \text{ or } pattern[j-s] \neq pattern[j]) \text{ and} \\ & ((s \geq i \text{ or } pattern[i-s] = pattern[i]) \text{ for } j < i \leq m)\} . \end{aligned}$$

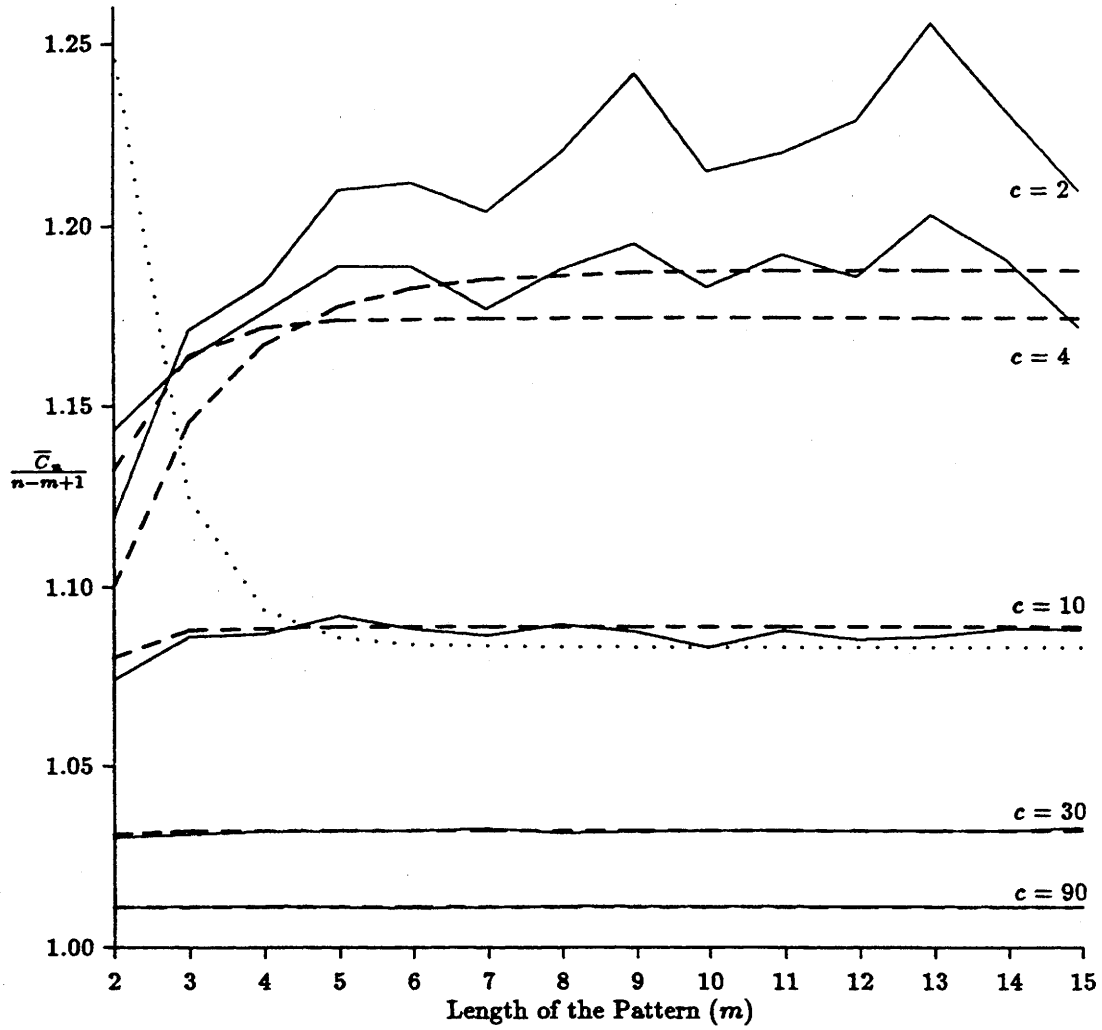

Figure 4.9: Simulation results for the Knuth-Morris-Pratt algorithm in random text. (The dashed line is the theoretical approximation, and the dotted line is Barth's approximation for $c = 4$.)

The *occurrence* heuristic is obtained by noting that we must align the position in the text that caused the mismatch with the first character of the pattern that matches it. Formally calling this table d, we have

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

for every symbol x in the alphabet. See Figure 4.10 for the code to compute both tables ($\widehat{dd}$ and d) from the pattern.

```
bmsearch( text, n, pat, m )
char text[], pat[];
int n, m;
{
    int k, j, skip;
    int dd[MAX_PATTERN_SIZE], d[MAX_ALPHABET_SIZE];

    initd( pat, m, d );     /* Preprocess the pattern */
    initdd( pat, m, dd );
    k = m; skip = dd[1] + 1;
    while( k <= n )
    {
        j = m;
        while( j>0 && text[k] == pat[j] )
        {
            j--; k--;
        }
        if( j == 0 )
        {
            Report_match_at_position( k+1 );
            k += skip;
        }
        else k += max( d[text[k]], dd[j] );
    }
}
```

Figure 4.10: The Boyer-Moore algorithm.

Both shifts can be precomputed based solely on the pattern and the alphabet. Hence, the space needed is $m + c + O(1)$. Given these two shift functions, the algorithm chooses the larger one. The same shift strategy can be applied after a match. In Knuth *et al* [KMP77] the preprocessing of the pattern is shown to be linear in the size of the pattern, as it is for the KMP algorithm; however, their algorithm is incorrect. The corrected version can be found on Rytter's paper [Ryt80]; see Figure 4.11.

```
initd( pat, m, d )  /* Preprocess pattern of length m : d table */
char pat[];
int m, d[];
{
    int k;

    for( k=0; k <= MAX_ALPHABET_SIZE; k++ ) d[k] = m;
    for( k=1; k<=m; k++ ) d[pat[k]] = m-k;
}

initdd( pat, m, d )  /* Preprocess pattern of length m : dd hat table */
char pat[];
int m, dd[];
{
    int j, k, t, t1, q, q1;
    int f[MAX_PATTERN_SIZE+1];

    for( k=1; k<=m; k++ ) dd[k] = 2*m-k;
    for( j=m, t=m+1; j > 0; j--, t-- )       /* setup the dd hat table */
    {
        f[j] = t;
        while( t <= m && pat[j] != pat[t] )
        {
            dd[t] = min( dd[t], m-j );
            t = f[t];
        }
    }
    q = t; t = m + 1 - q; q1 = 1; /* Rytter correction */
    for( j=1, t1=0; j <= t; t1++, j++ )
    {
        f[j] = t1;
        while( t1 >= 1 && pat[j] != pat[t1] ) t1 = f[t1];
    }
    while( q < m )
    {
        for( k=q1; k<=q; k++ ) dd[k] = min( dd[k], m+q-k );
        q1 = q + 1; q = q + t - f[t]; t = f[t];
    }
}
```

Figure 4.11: Preprocessing of the pattern in the Boyer-Moore algorithm.

Knuth *et al* [KMP77] have shown that, in the worst case, the number of comparisons is $O(n + rm)$, where r is the total number of matches. Hence, this algorithm can be as bad as the naive algorithm when we have many matches, namely, $\Omega(n)$ matches. A simpler alternative proof can be found in a paper by Guibas and Odlyzko [GO80]. Our simulation results agree well with the empirical and theoretical results in the original paper [BM77].

For a given fixed pattern, let $Pr\{head\}_k$ be the probability that position k in the text is the starting point (or *head*) for a right to left comparison. Position k is potentially a head, if position $k - j$ was a head and we did not shift more than j characters in that position. Thus, we obtain the following recurrence [BYR88],

$$Pr\{head\}_k = 1 - \sum_{j=1}^{m-1} Pr\{shift > j\} Pr\{head\}_{k-j} ,$$

with initial conditions $Pr\{head\}_m = 1$ and $Pr\{head\}_k = 0$, for $k < m$. Note that if in position $k - j$ we had a shift of less than j, say i, that case is considered now in position $k - j + i$ (that is, a different value of j).

It is not difficult to prove [BYR88] that this recurrence converges to

$$Pr\{head\}_\infty = \frac{1}{1 + \sum_{j=1}^{m-1} Pr\{shift > j\}} = \frac{1}{E[shift]} .$$

Hence, the average $Pr\{head\}$ over all possible patterns is

$$Pr\{head\} = E_{all\ patterns}\left[\frac{1}{E[shift]}\right] .$$

We use the Kantorovich inequality to bound the above expression by

$$Pr\{head\} \geq \frac{1}{E_{all\ patterns}[E[shift]]} .$$

A different approach using generating functions to enumerate all possible patterns is pursued in Baeza-Yates and Regnier [BYR88].

Suppose that a mismatch occurs at position $k + 1$ from the right of the pattern. Then, the shift due to the occurrence heuristic is $m - k$ if the mismatched character does not appear in the pattern; or is j ($j = 1, \ldots, m - k - 1$) if the mismatched character appears j positions to the left and does not appear to the right of that position; or is one, if the mismatched character appears in one of the k positions at the right. Hence, given a mismatch at position $k + 1$ from the right, the probability that the shift (due to the occurrence table) is j is

$$O_j(k) = \begin{cases} 1 & j = 1, k = m - 1 \\ 1 - (1 - 1/c)^{k+1} & j = 1, k < m - 1 \\ (1 - 1/c)^{k+j-1}/c & 1 < j < m - k \\ (1 - 1/c)^{m-1} & j = m - k \\ 0 & m - k < j \leq m . \end{cases}$$

In Boyer-Moore's original version [BM77], given a mismatch at position $k+1$ from the right, the shift for the match heuristic is of size k, if the matched substring also matches the pattern k positions to the left (if $j \geq m-k$ the corresponding suffix of the substring matches a prefix of the pattern) and does not match anywhere to the right of that position. Hence, the probability that the shift due to the match table is j given a mismatch at position $k+1$ from the right is defined recursively as [BM77]

$$M_j(k) = a_j(k)(1 - \sum_{i=1}^{j-1} M_i(k)) \ , \quad M_1(k) = a_1(k) \ ,$$

with

$$a_j(k) = \begin{cases} (1-1/c)(1/c)^k & 1 \leq j < m-k \ . \\ (1/c)^{m-j} & m-k \leq j \leq m \ . \end{cases}$$

The solution of this recurrence is

$$M_j(k) = a_j(k) \prod_{i=1}^{j-1} (1 - a_i(k)) \ , \quad j > 1 \ .$$

The probability that the maximum of the two tables is j, given a mismatch at position $k+1$ from the right, is

$$P_j(k) = O_j(k) \sum_{i=1}^{j} M_i(k) \ + \ M_j(k) \sum_{i=1}^{j-1} O_i(k) \ .$$

Hence, the average shift is

$$\overline{S} = \sum_{k=0}^{m-1} \frac{(1-1/c)}{c^k} \sum_{j=1}^{m} j P_j(k) + \frac{1}{c^m} \ .$$

The expected number of comparisons per trial is at least $\overline{C}_m$ (Lemma 4.2.3). If the previous shift was due to the occurrence heuristic and was less than $m-k$, then we know that in the first position from the right we have two characters that match. On the other hand, if the previous shift was due to the match heuristic and its value is s, then we know that $\min(m-s, k)$ characters match (common substring). Hence,

$$\overline{C}_n \geq \frac{\overline{C}_m}{\overline{S}}(n-m+1)$$

because the expected number of trials is bounded by $(n-m+1)/\overline{S}$ from the Kantorovich inequality. Figure 4.12 compares the empirical results with our theoretical results.

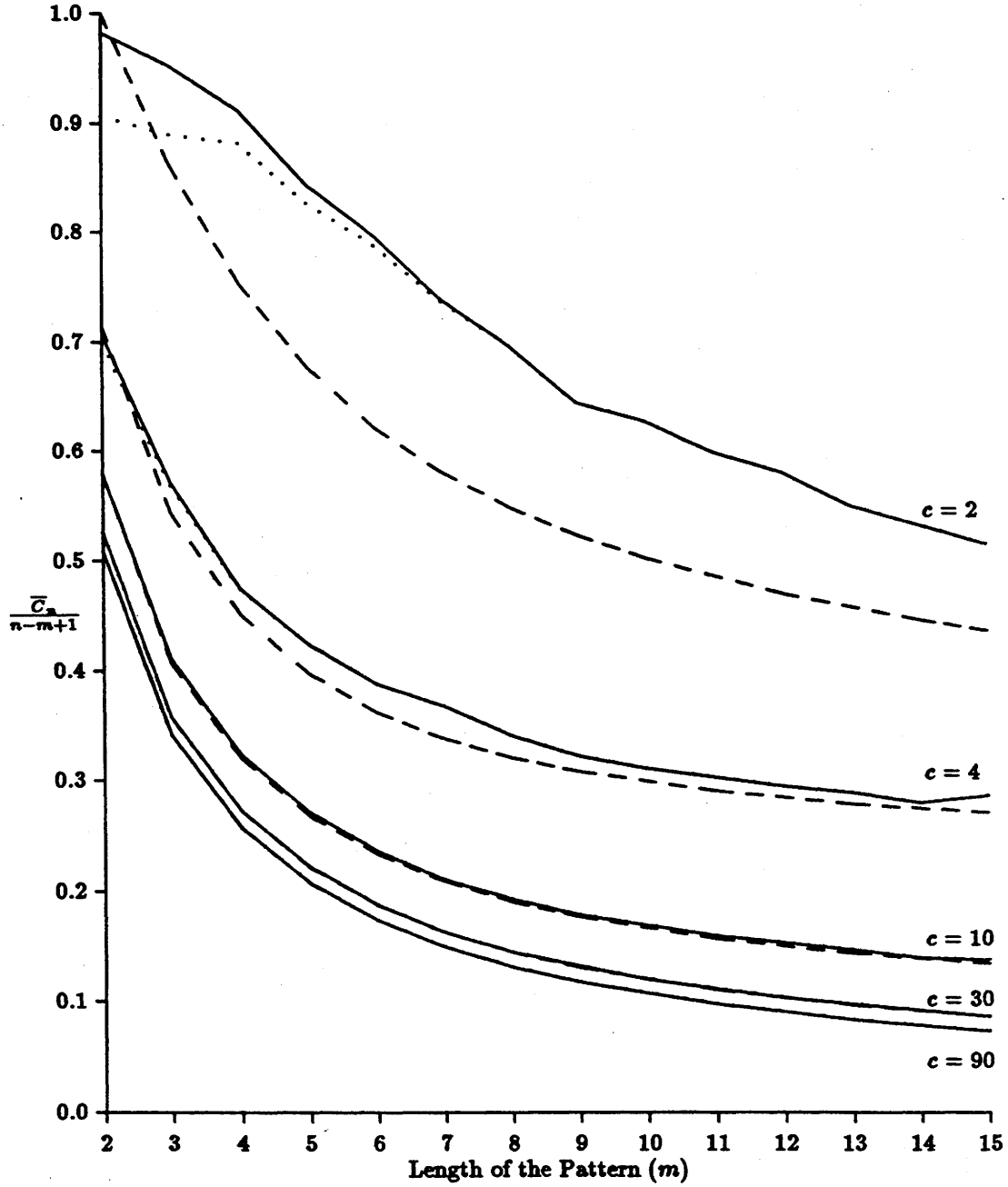

Figure 4.12: Simulation results for the Boyer-Moore algorithm in random text.
(The dotted lines shows Galil's version, and the dashed lines show the theoretical results.)

We are not able to compute a closed form for the previous expression, mainly because $M_j(k)$ is not in a closed form. We have obtained, however, the following asymptotic result for a large alphabet of size c, namely,

$$\frac{\overline{C}_n}{n-m+1} \geq \frac{1}{m} + \frac{m(m+1)}{2m^2c} + O(c^{-2}) .$$

To improve the worst case, Galil [Gal79] modifies the algorithm, so that it remembers how many overlapping characters it can have between two successive matches. That is, we compute the length, ℓ, of the longest prefix that is also a suffix of the pattern. Then, instead of going from m to 1 in the comparison loop, the algorithm goes from m to k, where $k = \ell - 1$ if the last event was a match, or $k = 1$ otherwise. For example, $\ell = 3$ for the pattern *ababa*. This algorithm is truly linear, with a worst case of $O(n+m)$ comparisons. However, according to empirical results (the dotted lines in Figure 4.12 are overlapped with the solid lines for $c > 2$), as expected, it only improves the average case for small alphabets, at the cost of using more instructions. Recently, Apostolico and Giancarlo [AG86] improved this algorithm to a worst case of $2n - m + 1$ comparisons.

4.6.1 The Simplified Boyer-Moore Algorithm

A simplified version of the Boyer-Moore algorithm (simplified-Boyer-Moore or SBM algorithm) is obtained by using only the *occurrence* heuristic. The main reason behind this simplification is that in practice patterns are not periodic. Also, the extra space needed decreases from $O(m + c)$ to $O(c)$. That is, the space depends only on the size of the alphabet (almost always fixed) and not on the length of the pattern (variable). For the same reason, it does not make sense to write a simplified version which uses Galil's improvement, because we need $O(m)$ space to compute the the length of the overlapping characters. Of course the worst case is now $O(mn)$, but it will be faster on the average.

Lemma 4.6.1 *The expected value of each shift of the pattern over the text using the simplified Boyer-Moore algorithm is*

$$\overline{S} = \frac{c^3 - c^2 + 1}{c^2 - c + 1} - c\left(1 - \frac{1}{c}\right)^m - \frac{c-1}{c^m} + \frac{c^2}{c^2 - c + 1}\left(\frac{c-1}{c^2}\right)^m .$$

Proof: We take the average of the occurrence table over all possible mismatches, that is

$$\overline{S} = \frac{1}{c^m} + \sum_{k=0}^{m-1}(1 - 1/c)(1/c)^k \sum_{j=1}^{m}(j \times O_j(k)) ,$$

giving the desired result. ■

Lemma 4.6.2 *The expected number of comparisons in each trial of the SBM algorithm given the last shift is*

$$\overline{C}'_m = \frac{c}{c-1}\left(1-\frac{1}{c^m}\right) - \frac{c}{c^2-c+1}\left[c(c-1)\left(\frac{1}{c^3-c+1}-\frac{1}{c^m}\right)+\left(\frac{c-1}{c^2}\right)^m - \frac{c^4}{c^3-c+1}\left(\frac{c-1}{c^3}\right)^m\right].$$

Proof: After a shift, if the shift was because the character that mismatched was in the pattern to the left of the mismatched position (and not to the right), then we know information about one character. In fact, if the shift was j, and the position of the mismatch was $k+1$ from the right, then we know that the comparison at position $k+j+1$ is a match. The probability of having a shift of size j of this kind after a mismatch in position $k+1$ is

$$O'_j(k) = \frac{(1-1/c)^{k+j-1}}{c},$$

for $1 \leq j < m-k$. In all other cases the number of comparisons is $\overline{C}_m$ (Lemma 4.2.3). Thus, we have

$$\overline{C}'_m = \overline{C}_m + \sum_{k=0}^{m-1} \frac{(1-1/c)}{c^k} \sum_{j=1}^{m-k-1} \left(\frac{1}{c^{k+j}} - \frac{1}{c^{m-1}}\right) O'_j(k).$$

■

THEOREM **4.6.1** *The expected number of text-pattern comparisons performed by the simplified Boyer-Moore algorithm to search with a pattern of length m in a text of length n $(n \geq m)$ is*

$$\overline{C}_n \geq \frac{\overline{C}'_m}{\overline{S}}(n-m+1).$$

Asymptotically for n and m (with $m \ll n$) it is

$$\frac{\overline{C}_n}{n-m+1} \geq \frac{1}{c-1} + \frac{1}{c^4} + O(c^{-6})$$

and asymptotically for n and c (with $c \ll n$) it is

$$\frac{\overline{C}_n}{n-m+1} \geq \frac{1}{m} + \frac{m^2+1}{2cm^2} + O(c^{-2}).$$

Proof: By using the Kantorovich inequality, the expected number of trials is bounded by $(n-m+1)/\overline{S}$ (Lemma 4.6.1). In each trial, we perform $\overline{C}'_m$ comparisons (Lemma 4.6.2). ■

Figure 4.13 shows the theoretical and empirical results for this case.

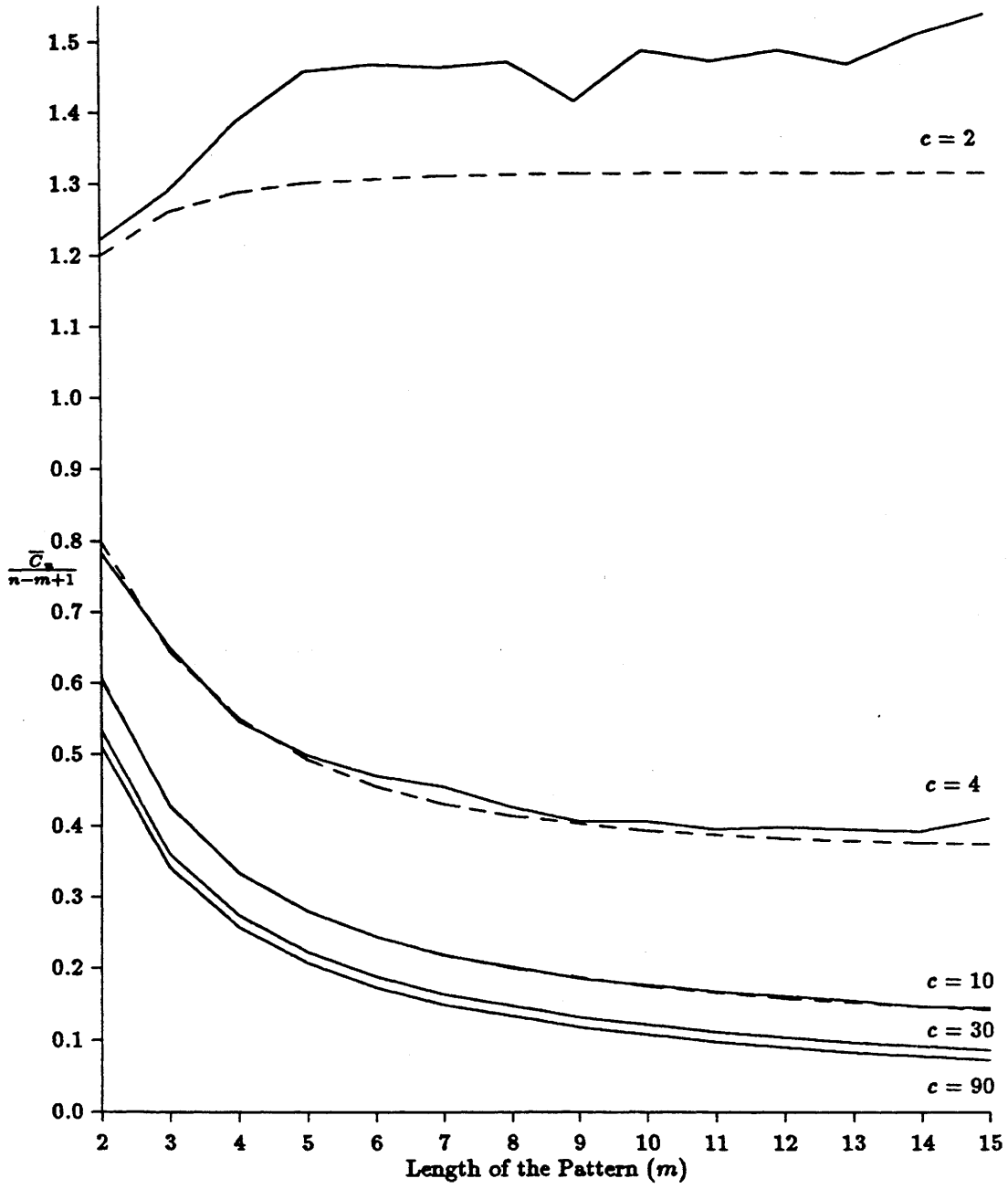

Figure 4.13: Simulation results for the simplified Boyer-Moore algorithm in random text.

(The dashed line is the theoretical lower bound.)

4.6.2 The Boyer-Moore-Horspool Algorithm

An interesting observation that leads to an improvement of the SBM algorithm, is that when we know that the pattern either matches or does not, any of the characters from the text can be used to address the heuristic table. Based on this, Horspool [Hor80] improved the SBM algorithm, by addressing the occurrence table with the character in the text corresponding to the last character of the pattern. We call this algorithm, the Boyer-Moore-Horspool or BMH algorithm.

To avoid a comparison in the case that the value in the table is zero (the last character of the pattern), we define the initial value of the entry in the occurrence table corresponding to the last character in the pattern as m and then we compute the occurrence heuristic table for only the first $m-1$ characters of the pattern. Formally

$$d[x] = \min\{s | s = m \text{ or } (1 \leq s < m \text{ and } pattern[m-s] = x)\} .$$

The code for an efficient version of the Boyer-Moore-Horspool algorithm is extremely simple and is presented in Figure 4.14 where `MAX_ALPHABET_SIZE` is the size of the alphabet.

Lemma 4.6.3 *The expected value of a shift of the pattern over the text for the BMH algorithm is*

$$\overline{S} = c\left(1 - \left(1 - \frac{1}{c}\right)^m\right) .$$

Corollary 4.6.1 *For any probability distribution of the alphabet,* $\overline{S} > 1$.

Proof: The value of the shift is m if the character in the text is different from any character in the pattern between positions 1 and $m-1$ inclusive. The probability of that event is $(1-1/c)^{m-1}$. The value of the shift is $m-j$ if the character of the text is equal to p_j and is different from all the characters in the pattern between positions $j+1$ and $m-1$ inclusive. The probability of this case is $(1/c)(1-1/c)^{m-j-1}$. Thus,

$$\overline{S} = m(1-1/c)^{m-1} + 1/c \sum_{j=1}^{m-1} (m-j)(1-1/c)^{m-j-1} = c(1-(1-1/c)^m) .$$

For a non-uniform distribution, we replace c by $1/p_{equal}$, and $(1-p_{equal})^{m-1} < 1$ implies $\overline{S} > 1$. This is always true for $c > 1$ and $m > 1$. ■

Lemma 4.6.4 *The expected number of comparisons per trial for the Boyer-Moore-Horspool algorithm is*

$$\overline{C}'_m = \frac{c}{c-1}\left(1 - \frac{1}{c^{m-1}}\right) + \frac{1}{c^2-c+1}\left(1 + c^3\left(\frac{c-1}{c^2}\right)^m\right) .$$

```
bmhsearch( text, n, pat, m )   /* Search pat[1..m] in text[1..n] */
char text[], pat[];
int n, m;
{
    int d[MAX_ALPHABET_SIZE], i, j, k;

    for( j=0; j<MAX_ALPHABET_SIZE; j++ ) d[j] = m; /* preprocessing */
    for( j=1; j<m; j++ ) d[pat[j]] = m-j;
    pat[0] = CHARACTER_NOT_IN_THE_TEXT;     /* to avoid having code */
    text[0] = CHARACTER_NOT_IN_THE_PATTERN; /* for special cases    */

    i = m;
    while( i <= n )                         /* search               */
    {
        k = i;
        for( j=m; text[k] == pat[j]; j-- ) k--;
        if( j == 0 ) Report_match_at_position( k+1 );
        i += d[text[i]];
    }
    /* restore pat[0] and text[0] if necessary */
}
```

Figure 4.14: The Boyer-Moore-Horspool algorithm.

Proof: After a shift, we know that if the shift was less than m we have a position where the character in the text matches the character in the pattern. If the last shift was j, then the character in position $j+1$ of the pattern from the right matches the character in the text. Therefore, the expected number of comparisons depending on the last shift is

$$\overline{C}'_m = Prob\{shift = m\}\overline{C}_m \; + \; \sum_{j=1}^{m-1} Prob\{shift = j\} \left(\overline{C}_{m-1} + \frac{1}{c^j} \right) .$$

Using the probabilities given in the previous lemma and Lemma 4.2.3 we obtain the desired answer. ■

THEOREM **4.6.2** *The expected number of text-pattern comparisons performed by the Boyer-Moore-Horspool algorithm to search with a pattern of length m in a text of length n ($n \geq m$) is*

$$\overline{C}_n \geq \frac{\overline{C}'_m}{\overline{S}}(n - m + 1) .$$

Asymptotically for n and m (with $m \ll n$) it is

$$\frac{\overline{C}_n}{n-m+1} \geq \frac{1}{c-1} + O((1-1/c)^m) ,$$

and asymptotically for n and c (with $c \ll n$ and $m > 4$) it is

$$\frac{\overline{C}_n}{n-m+1} = \frac{1}{m} + \frac{m+1}{2mc} + O(c^{-2}) .$$

Proof: The expected number of times that we shift the pattern is bounded by $\lfloor (n-m+1)/\overline{S} \rfloor$ (Lemma 4.6.3). Each time, on the average, we perform $\overline{C}'_m$ comparisons (Lemma 4.6.4). The asymptotic result for n and c is an equality because by using the Kantorovich inequality we obtain

$$E[1/shift] \leq \left(1 + O\left(\frac{m^2}{c^2}\right)\right) \frac{1}{E[shift]}$$

because, for $m \leq c$, we have

$$m - \frac{m(m-1)}{2c} \leq shift \leq m - \frac{m-1}{c}$$

by taking a pattern with all the characters equal (largest shift), and a pattern with all the characters different (smallest shift). ■

Based on this analysis, the BMH algorithm is simpler and faster than the SBM algorithm (it is not difficult to prove that the expected shift is larger for the BMH algorithm), and is as good as the BM algorithm for alphabets of size at least 10. Figure 4.15 shows the empirical results and the theoretical results for Horspool's version.

Finally, Figure 4.16 shows the execution time of searching 1000 random patterns in random text for all the algorithms ($c = 30$). Based on the empirical results, it is clear, that Horspool's variant is the best known algorithm for almost all pattern lengths and alphabet sizes. Figure 4.17 shows the same empirical results of Figure 4.16, but for English text instead of random text. The results are similar, and improvements to the BMH algorithm for searching in English text are discussed in the next chapter.

4.7 Optimal Algorithms

The linear time worst case algorithms presented in previous sections are optimal in the worst case with respect to the number of comparisons[Riv77]; however, they are

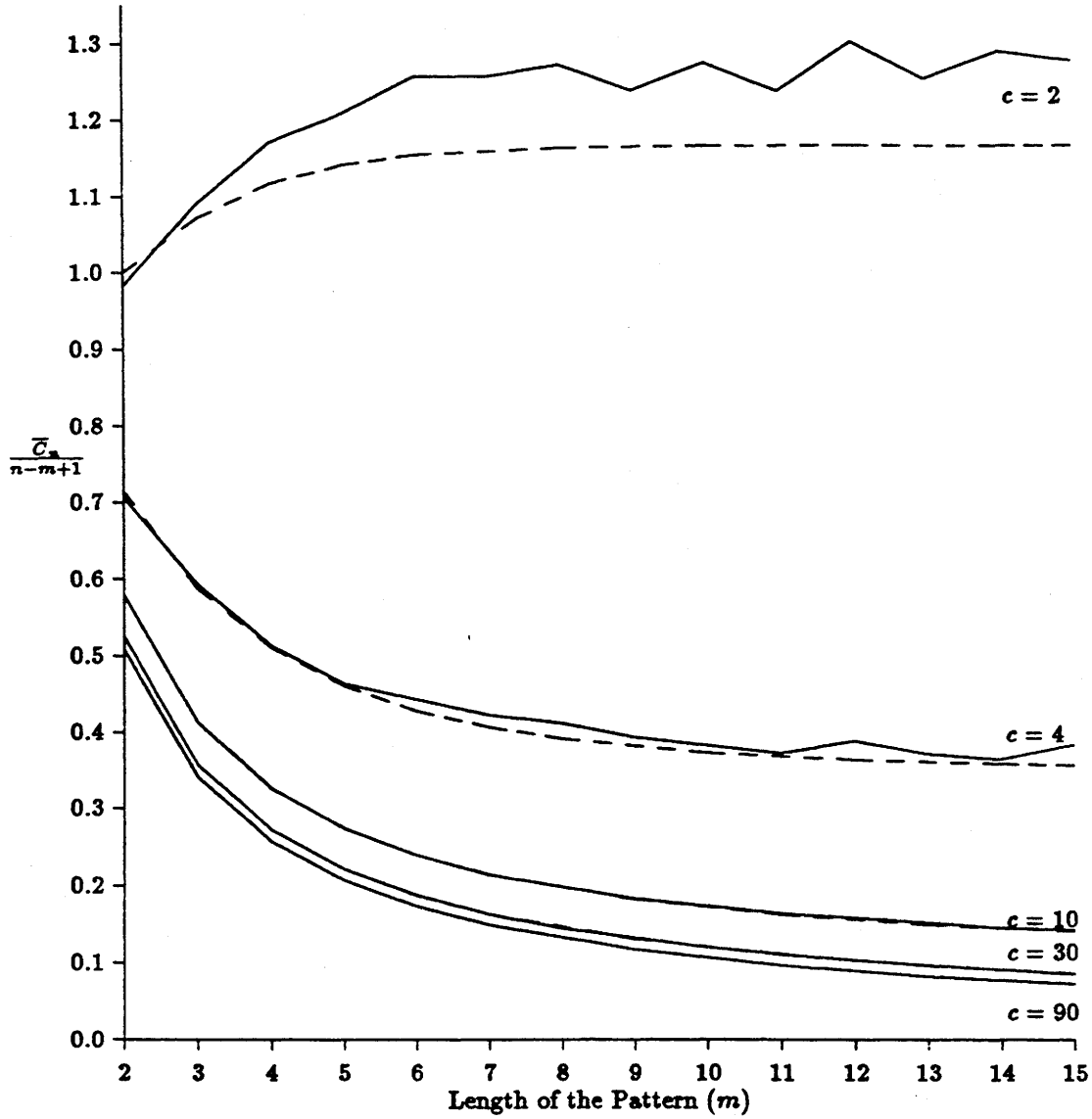

Figure 4.15: Simulation results for the Boyer-Moore-Horspool algorithm in random text.

(The dashed line is the theoretical lower bound.)

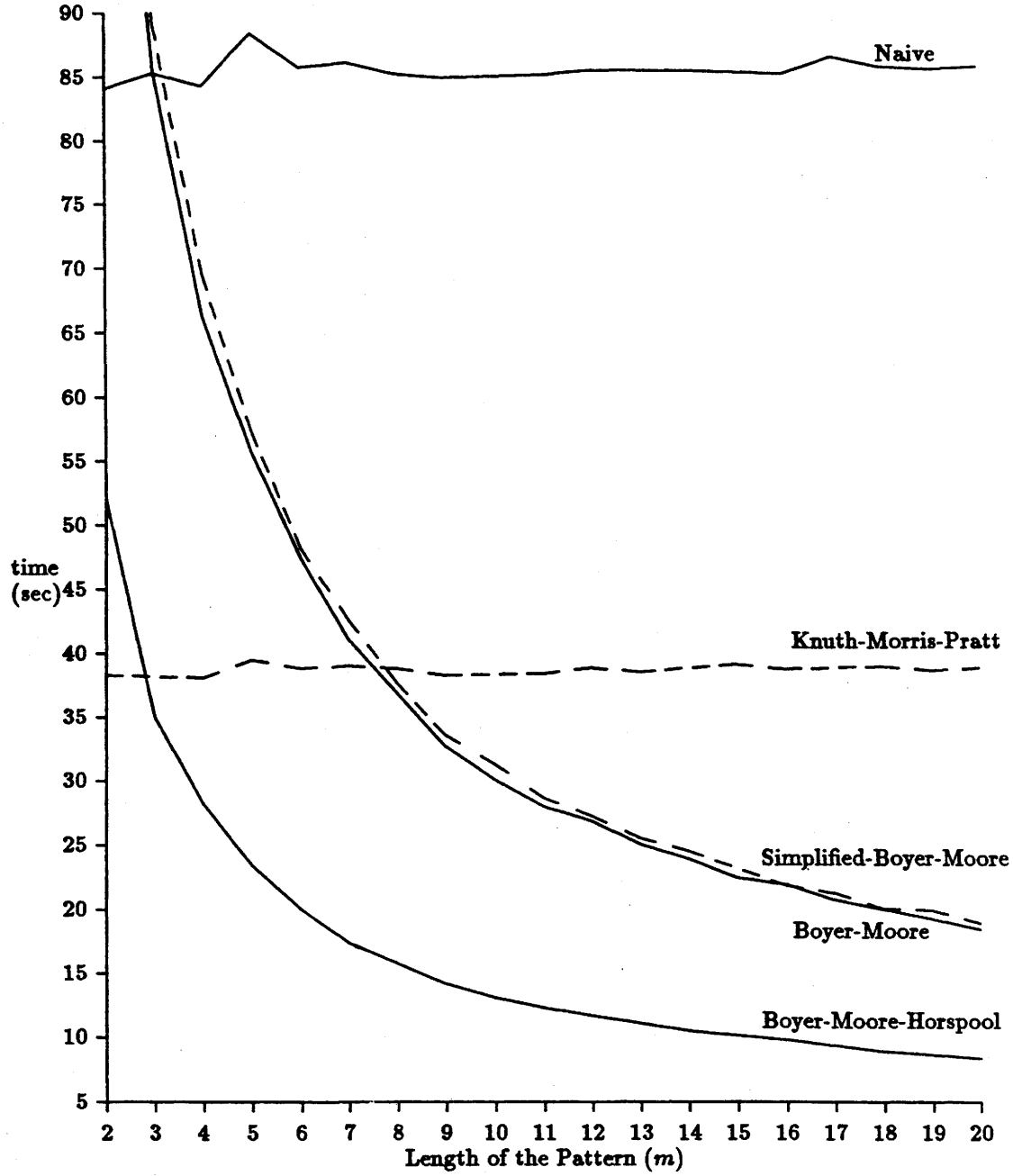

Figure 4.16: Simulation results for all the algorithms in random text ($c = 30$).

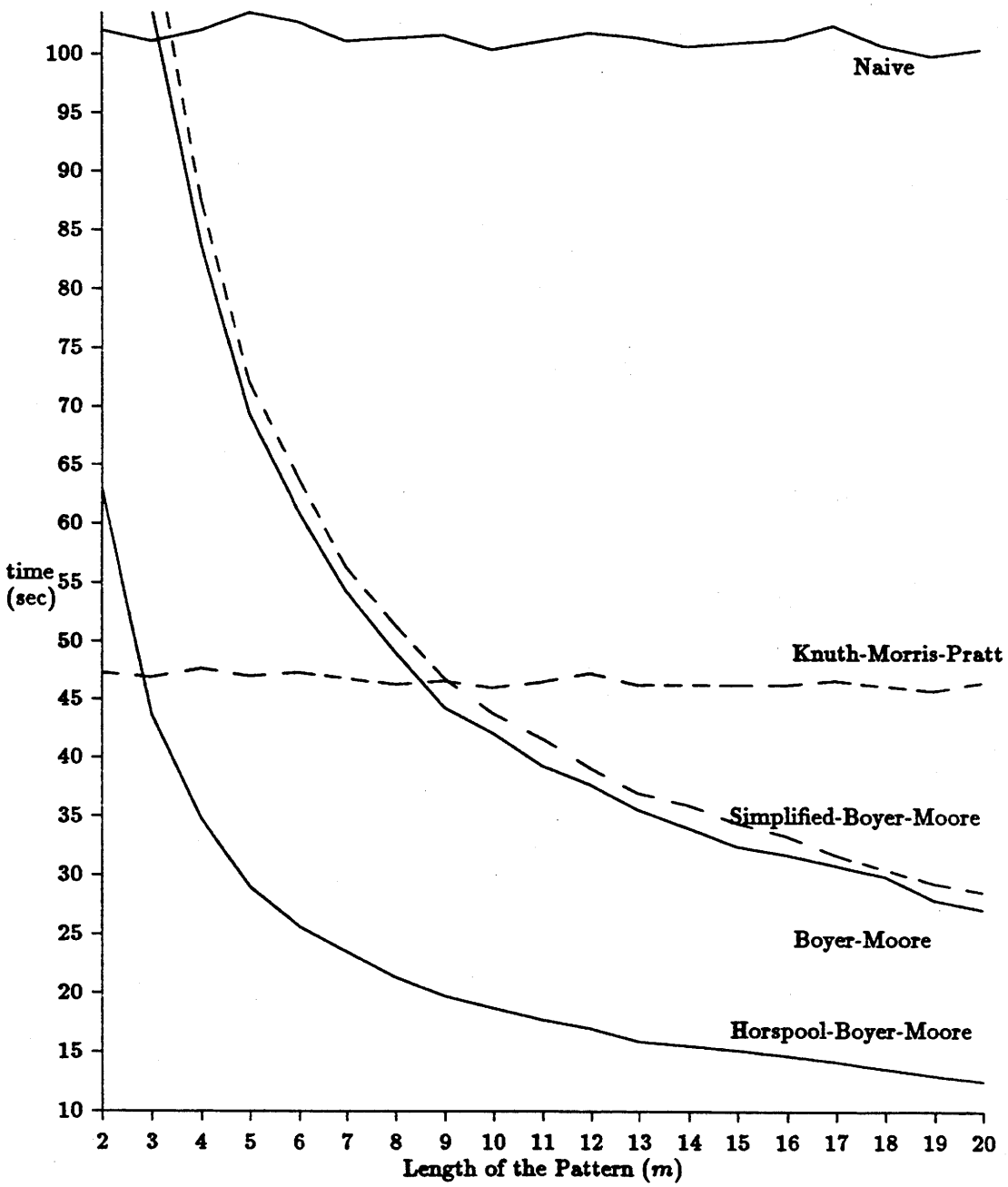

Figure 4.17: Simulation results for all the algorithms in English text.

not space optimal in the worst case, because they use space that depends on the size of the pattern, the size of the alphabet, or both. Galil and Seiferas [GS80, GS83] show that is possible to have linear time worst case algorithms using constant space. Also they show that the delay between reading two characters of the text is bounded by a constant (this may be interesting for real time searching algorithms) [Gal81]. Practical algorithms that achieve optimal worst case time and space are presented in [CP88, Cro88, CP89].

Yao's result [Yao79] provides a lower bound for the average number of inspections. What happens with the average case for the number of comparisons? First, we present the optimal algorithm for the number of inspections for a pattern of length two [KMP77] using Markov chains. Figure 4.18 shows the Markov chains for the two possible patterns (two equal characters "aa" or two different characters "ab"). Note that the pattern is scanned right to left, and that we have two labels on each edge. One is the value of the last character inspected, and the second (in parentheses) is how many positions we shift the pattern. When the shift is zero, it means the first character matched, and that we have to compare the next character. The labels in each state indicate how much information we know about the two characters in the text corresponding to the pattern, before inspecting a new character ("?" means that the value of the character is unknown) and the number of the state. The algorithm is optimal on average in the sense that it uses all the information from the past, and then it inspects the minimum possible number of characters.

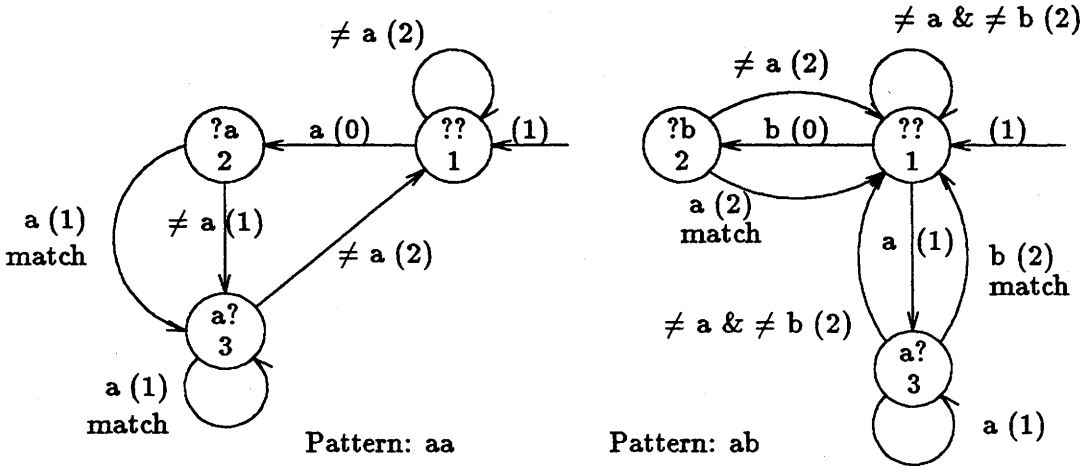

Figure 4.18: Optimal algorithms for a pattern of length two.

For the pattern "aa" we have

$$\mathbf{P} = \begin{bmatrix} 1-1/c & 1/c & 0 \\ 0 & 0 & 1 \\ 1-1/c & 0 & 1/c \end{bmatrix},$$

because the probability of matching a character is $1/c$. Multiple edges going from one state to another state are collapsed into one edge having the sum of their probabilities. Therefore,

$$\pi = \frac{1}{c^2+c-1}[\, c(c-1),\ c-1,\ c\,] .$$

Now, we can compute the expected number of characters shifted for each inspection, or *csi* for short (for each transition in which we advance the text, we inspect a new character). For each state, we multiply the probability of each transition by the corresponding shift and by the stationary state probability; that is,

$$E[csi] = 2(1-1/c)p_1 + p_2 + (1/c + 2(1-1/c))p_3 = \frac{c(2c-1)}{c^2+c-1} .$$

$E[csi]$ for the pattern "ab" is equal to $E[csi]$ for the pattern "aa" (although the Markov chains are different). Thus, noting that we have one inspection for each state visited, we have the result of Knuth *et al* [KMP77] for the expected number of inspections:

$$\overline{I}_n = \frac{1}{E[csi]} = \frac{c^2+c-1}{c(2c-1)}n + O(1) .$$

This result is optimal in the number of inspections [KMP77]. The expected number of comparisons ($\overline{C}_n$) of this algorithm is greater than $\overline{I}_n$. The reason is that for the case "ab," in state 1, we need two comparisons to decide which is the new state (three-way branch) and in state 3, we need another comparison to decide if we should report a match (unless $c = 2$). Arranging the comparisons to minimize the number of times that the second comparison is performed and using different code for each case, we have an algorithm that performs fewer comparisons on the average than the algorithm given by Knuth *et al* [KMP77] (see Figure 4.18). Table 4.2 shows the empirical results for both algorithms.

Lemma 4.7.1 *The expected number of comparisons performed by the previous algorithm to search with a pattern of length two in a text of length n is bounded by*

$$\frac{2c(c^2+c-1)}{2c^3+3c^2-4c+1} \le \frac{\overline{C}_n}{n} \le \frac{9c(c^2+c-1)}{4(2c^3+3c^2-4c+1)} .$$

Algorithm	$c = 2$	$c = 4$	$c = 10$	$c = 30$	$c = 90$
Knuth *et al* [KMP77]	1.1716	1.1086	1.04762	1.01654	1.00551
Ours	0.969	0.972	0.983	0.997	1.0005

Table 4.2: $\overline{C}_n/n$ for optimal algorithms ($m = 2$).

Proof: Adding the expected number of characters shifted per comparison performed by the algorithm in each state, weighted by the probability of the state, and using the Kantorovich inequality (C_n/n ranges from 1 to 2). ■

Is the previous algorithm optimal with respect to the number of comparisons? The answer is: No. An algorithm that behaves like the previous algorithm for the pattern "aa", and like the Boyer-Moore algorithm (presented in Section 4.6) for the pattern "ab" performs fewer comparisons on the average, for all $c > 2$. In fact, this algorithm is Galil's improvement of the Boyer-Moore algorithm [Gal79]. In this case, the expected number of comparisons is

$$\frac{\overline{C}_n}{n} = \frac{c^3 + c^2 - 1}{c^2(2c-1)} + O(1/n) < 1 .$$

This may seem counterintuitive. What happens is that we are replacing some comparisons by table lookups (for a finite alphabet). From these results we conjecture that in general $\overline{C}_n > \overline{I}_n$, and that optimal algorithms for each of the measures $\overline{I}_n$ and $\overline{C}_n$ are different.

Using the same approach it is possible to obtain $\overline{C}_n$ for any algorithm, for a pattern of small fixed length. Table 4.3 shows the results for patterns of length two as a function of the alphabet size c for Boyer-Moore type algorithms (see Section 4.6).

Algorithm	$\overline{C}_n/n + O(1/n)$
Boyer-Moore [BM77, KMP77]	$\frac{c+1}{2c-1}$
Boyer-Moore-Galil [Gal79]	$\frac{c^3+c^2-1}{c^2(2c-1)}$
Simplified Boyer-Moore [BM77]	$\frac{(2c+1)(c+1)}{2c(2c-1)}$
Boyer-Moore-Horspool [Hor80]	$\frac{c+1}{2c-1}$

Table 4.3: $\overline{C}_n/(n-1)$ for Boyer-Moore type algorithms ($m = 2$).

An interesting open problem is to determine the number of states in a deterministic finite automaton that represents an optimal algorithm that scans the pattern

right to left (or a "Boyer-Moore DFA"). An obvious upper bound is 2^m. Knuth *et al.* [KMP77] explain why a pattern consisting of m different symbols requires $O(m^2)$ states (basically, all possible substrings in a pattern of length m). An $O(m^3)$ lower bound for this problem restricted to a three character alphabet is known [Gal85].

4.8 The Karp-Rabin Algorithm

A different approach to string searching is to use hashing techniques. All that we need to do is to compute the signature function of each possible m-character substring in the text and check if it is equal to the signature function of the pattern.

Karp and Rabin [KR87] found an easy way to compute these signature functions efficiently for the signature function $h(k) = k \bmod q$, where q is a large prime. Their method is based on computing the signature function for position i given the value for position $i-1$. The algorithm requires time proportional to $n+m$ in almost all the cases, without using extra space. Note that this algorithm finds positions in the text that have the same signature value as the pattern, so, to ensure that there is a match, we must make a direct comparison of the substring with the pattern. This algorithm is probabilistic, but using a large value for q makes collisions unlikely (the probability of a random collision is $O(1/q)$).

Theoretically, this algorithm may still require mn steps in the worst case, if we check each potential match and have too many matches or collisions. In our empirical results we observed only 3 collisions in 10^7 computations of the signature function, for large alphabets.

The signature function represents a string as a base-d number, where $d = c$ is the number of possible characters. To obtain the signature value of the next position, only a constant number of operations are needed. The code for the case $d = 128$ (ASCII) and $q = 16647133$ based in Sedgewick's exposition [Sed83], for a word size of 32 bits, is given in Figure 4.19 (D= $\log_2 d$ and Q= q). By using a power of 2 for d ($d \geq c$), the multiplications by d can be computed as shifts. The prime q is chosen as large as possible, such that $(d+1)q$ does not cause overflow [Sed83]. We also impose the condition that d is a primitive root mod q. This implies that the signature function has maximal cycle; that is,

$$\min_k(d^k \equiv 1 \pmod q) \;=\; q-1\ .$$

Thus, the period of the signature function is much bigger than m for any practical case.

Karp and Rabin [KR87] show that

$$Prob\{collision\} \leq \frac{\pi(m(n-m+1))}{\pi(M)}\ ,$$

```
rksearch( text, n, pat, m )  /* Search pat[1..m] in text[1..n] */
char text[], pat[];          /* (0 < m <= n)                   */
int n, m;
{
    int h1, h2, dM, i, j;

    dM = 1;
    for( i=1; i<m; i++ ) dM = (dM << D) % Q; /* Compute the signature  */
    h1 = h2 = 0;                             /* of the pattern and of */
    for( i=1; i<=m; i++ )                    /* the beginning of the  */
    {                                        /* text                  */
         h1 = ((h1 << D) +  pat[i] ) % Q;
         h2 = ((h2 << D) +  text[i] ) % Q;
    }
    for( i = 1; i <= n-m+1; i++ )  /* Search */
    {
        if( h1 == h2 ) /* Potential match */
        {
            for(j=1; j<=m && text[i-1+j] == pat[j]; j++ ); /* check */
            if( j > m )                                    /* true match */
                Report_match_at_position( i );
        }
        h2 = (h2 + (Q << D) - text[i]*dM ) % Q;  /* update the signature */
        h2 = ((h2 << D) +  text[i+m] ) % Q;      /* of the text          */
    }
}
```

Figure 4.19: The Karp-Rabin algorithm.

where $\pi(x)$ is the number of primes less or equal to x, for $m(n-m+1) \geq 29$, $c = 2$, and q a random prime no greater than M. They use $M = O(mn^2)$ to obtain a probability of collision of $O(1/n)$. However, in practice, the bound M depends on the word size used for the arithmetic operations, and not on m or n. For this reason, our analysis follows a different approach, and is based on a more realistic model of computation.

Lemma 4.8.1 *The probability that two different random strings of the same length have the same signature value is*

$$Prob\{collision\} = \frac{1}{q} - O(\frac{1}{c^m}) < \frac{1}{q} ,$$

for a uniform signature function.

Proof: Let m be the length of the strings and c the alphabet size. There are c^m different strings of length m. Let $d = c$ be a primitive root mod q; than, the signature function is

$$h(s) = \left(\sum_{i=0}^{m-1} s_i c^i \right) \bmod q$$

where s_i is the i-th character of a string, and all possible values from 0 to $c^m - 1$ are possible before taking the modulus. Let $k = \lfloor c^m/q \rfloor$ be the number of times that q divides c^m. If we choose a string at random, we have two cases:

- with probability $p = (k+1)(c^m - qk)/c^m$ we have k possible signature values equal to the chosen one, and
- with probability $1 - p$ we only have $k - 1$ values that can be equal.

Thus, the probability of a collision is

$$\begin{aligned} Prob\{collision\} &= \frac{kp}{c^m - 1} + \frac{(k+1)(1-p)}{c^m - 1} \\ &= \frac{k(k+1)q}{c^m(c^m - 1)} \\ &= \frac{1}{q} - \frac{1}{c^m - 1} + O(q^{-1}c^{-m}) < \frac{1}{q} . \end{aligned}$$

■

THEOREM **4.8.1** *The expected number of text-pattern numerical comparisons performed by the Karp-Rabin algorithm to search with a pattern of length m in a text of length n is*

$$\frac{\overline{C}_n}{n} = \mathcal{H} + \frac{m}{c^m}\left(1 - \frac{1}{q}\right)\frac{1}{q} + O(1/c^m) ,$$

where $\mathcal{H}$ is the cost of computing the signature function expressed as comparisons.

Proof: We have that

$$\overline{C}_n = (n+m)\mathcal{H} + \frac{(n-m+1)m}{c^m} + (1 - \frac{1}{c^m})(n-m+1)m Prob\{collision\} + O(1) ,$$

where the first term is the cost of computing the signature values and the second term is the number of comparisons used to check all the matches found on average. For every position that does not match, we have to perform m comparisons if the two random strings have the same signature value. We bound this probability by using Lemma 4.8.1. ■

	c				
m	2	4	10	30	90
2	1.5	1.125	1.02	1.00222	1.00025
	1.4996	1.12454	1.01999	1.002179	1.000244
4	1.25	1.01563	1.00040	1.00000	1.00000
	1.2500	1.01565	1.000361	1.0000040	1.000
7	1.05469	1.00043	1.00000	1.00000	1.00000
	1.05422	1.000404	1.0000017	1.000	1.000
10	1.00977	1.00001	1.00000	1.00000	1.00000
	1.00980	1.0000050	1.000	1.000	1.000
15	1.00046	1.00000	1.00000	1.00000	1.00000
	1.000454	1.000	1.000	1.000	1.000

Table 4.4: Theoretical (top row) and experimental results for the Karp-Rabin algorithm.

The empirical results are compared with our theoretical results in Table 4.4 using $\mathcal{H} = 1$. They agree very well for all alphabet sizes. In practice, the value of $\mathcal{H}$ is bigger, due to the multiplications and the modulus operations of the algorithm. However, this algorithm becomes competitive for long patterns. We can avoid the computation of the modulus function at every step by using implicit modular arithmetic given by the hardware. In other words, we use the maximum value of an integer (word size) for q [Gon88a]. The value of d is selected such that $d^k \bmod 2^r$ has maximal cycle length (cycle of length 2^{r-2}), for r from 8 to 64, where r is the size, in bits, of a word. For example, an adequate value for d is 31.

With these changes, the evaluation of the signature function in every step (see Figure 4.19) is

```
h2 = h2*D - text[j-m]*dM + text[i+m]; /* update the signature value */
```

and overflow is ignored. In this way, we use two multiplications instead of one multiplication and two modulus operations.

Chapter 5

Improving the Boyer-Moore Algorithm

1969 J. ARGENTI *Managem. Techniques* 120
Improving the accuracy of a forecast usually calls
for the careful and detailed analysis of past records
plus an estimate or an inspired guess as to how
future trends will differ from those of the past.
OED2, **inspired guess**

In this chapter, we show that it is possible to improve the average time of the Boyer-Moore string matching algorithm by using additional space. This is accomplished by applying transformations that increase the size of the alphabet in use. This results in an algorithm that is more than 50% faster than the original one, for long patterns. We include experimental results on random and English text. Some improvements for searching English text, and some hybrid algorithms are also discussed; see also [BY89a, BY89c].

5.1 Introduction

Our starting point is Horspool's version of the Boyer-Moore algorithm (see Figure 4.14). From the previous chapter we have seen that in practice this is the fastest known string matching algorithm. Thus, a natural question is: Can it be improved? We will make use of a result obtained in Chapter 4. The expected value of each shift of the pattern over the text is

$$\overline{S}(m) = c\left(1 - \left(1 - \frac{1}{c}\right)^m\right) \approx m - \frac{m(m-1)}{2c} + O(c^{-2})$$

It is easy to see that if we increase c, then $\overline{S}(m)$ also increases. This result is also valid for a non-uniform distribution, if we replace c by $1/\sum_{j=1}^{c} p_j^2$ in $\overline{S}(m)$; where p_j is the probability of the j-th symbol of the alphabet.

5.2 Alphabet Transformations

From the analysis of the Boyer-Moore-Horspool algorithm, it is clear that a longer pattern or a larger alphabet improve the expected time. Since the pattern is given, it is not possible to increase its length. It is possible, however, to increase the size of the alphabet, which decreases the length of the pattern. One possible way (there are many of them) is to transform the pattern into a new pattern such that the i-th symbol is the concatenation of the i-th, $(i+1)$-st, ..., $(i+k-1)$-st characters of the original pattern, for some $k \geq 1$, where $k < m/2$. This transformation is very simple, and reduces the length of the pattern to $m-k+1$ in the new alphabet. On the other hand, the size of the alphabet is increased to c^k. This transformation is mentioned in Sedgewick [Sed83], where a value of $k \approx \ln(4m)$ is suggested and used for the SBM algorithm. Here, we find the optimal value of k for the Boyer-Moore-Horspool algorithm. A transformation of this kind is mentioned in Knuth *et al* [KMP77] for a theoretical algorithm of $O(n \log m/m)$ average running time. The idea is to look at the last block of b characters in the pattern (with $b = O(\log m)$) to compute the next shift. Variations of block-encoding are also discussed in Schaback [Sch88]. Our transformation is simpler, in the sense that it leads to a practical algorithm.

	k						
c	1		2		3	4	5
2	2-3	(2)	4-5	(3-)	6-7	8-11	12-15
4	2-3	(2-3)	4-8	(4-)	9-15	16-29	30-56
8	2-4	(2-5)	5-13	(6-)	14-36	37-100	101-
16	2-6	(2-6)	7-25	(7-)	26-96	97-	
32	2-8	(2-8)	9-47	(9-)	48-		
64	2-12	(2-12)	13-93	(13-)	94-		
128	2-16		17-184		185-		

Table 5.1: Range of pattern lengths such that k is optimal for some alphabets. (Experimental results within parentheses.)

For simplicity we discuss the application of this transformation only to the occurrence heuristic. Note that this transformation also reduces the probability of having longer periodicities in the pattern. Using Lemma 4.6.3 the expected shift is

now

$$\overline{S} = c^k(1 - (1 - 1/c^k)^{m-k+1}) ,$$

and we want to find an integer value of k that maximizes this expression. The solution for real k is given by the following transcendental equation

$$1 = (1 - 1/y)^{m-k+1} \left(1 + \frac{m-k+1}{y-1} - \log_c(1 - 1/y)\right) ,$$

where $y = c^k$. If k^* is its solution, we are interested in either $\lfloor k^* \rfloor$ or $\lceil k^* \rceil$. Table 5.1 shows the range of pattern lengths such that a value of k is optimal for different values of c. Figure 5.1 shows graphically the optimal values of k by regions, for any value of c and m in the range from 2 to 40. For large values of m and c, we have $k = O(\log_c m)$. Table 5.2 shows the ratio between the expected shift length of the original algorithm and the for optimal k, for some values of c and m.

	m								
c	4		8		16		32	64	128
2	.8108	(.7856)	.4515	(.5824)	.1965	(.5511)			
4	.9709	(.9537)	.6189	(.5801)	.3118	(.3655)			
8	1.0000	(1.1076)	.7861	(.7723)	.5104	(.5050)	.2705		
16	1.0000	(1.2149)	.9326	(.9264)	.7058	(.6944)	.4674		
32	1.0000	(1.2723)	1.0000	(1.0256)	.8555	(.8523)	.6682	.4489	.2501
64	1.0000	(1.3024)	1.0000	(1.0828)	.9520	(.9516)	.8203	.6500	.4404
128	1.0000		1.0000		1.0000		.9173	.8034	.6410

Table 5.2: Ratio between $\overline{S}$ for optimal k and $k = 1$.
(Experimental results between parentheses.)

5.3 Implementation and Experimental Results

The transformation is equivalent to asking where in the pattern k characters of the text appear; that is, we are indexing the occurrence table with a substring of k characters. Clearly, the implementation of this transformation is most suitable for alphabets that are powers of 2, because in this case we can compute the index value of a substring using shift operations (instead of multiplications that are too expensive for practical purposes). If the alphabet size is not a power of 2, we can always use an alphabet of size $2^{\lceil \log_2 c \rceil}$. Note that it is not necessary to change the code for the actual comparison between the string and the text. For example, if $k = 2$ only four lines are changed in the Boyer-Moore-Horspool algorithm (Figure 4.14) as

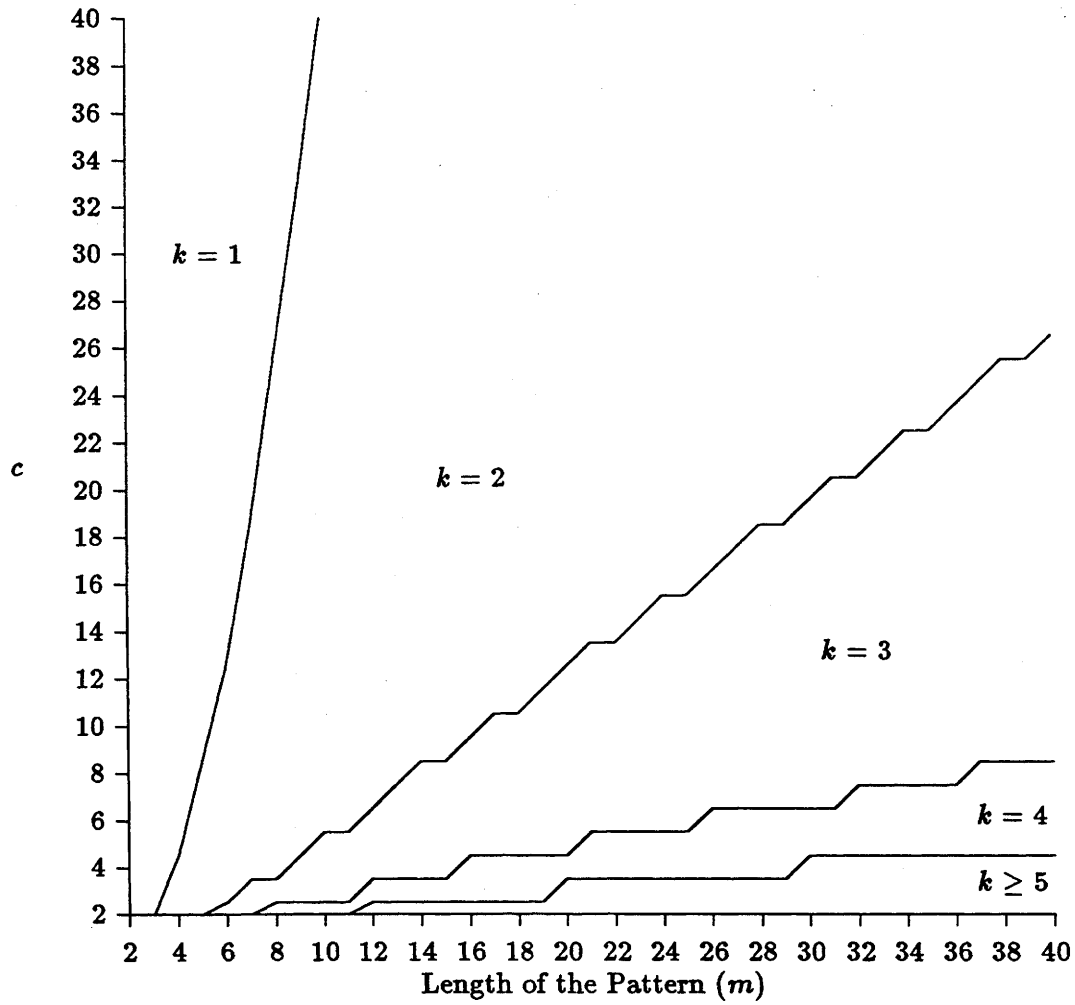

Figure 5.1: Optimal value of k (regions) for any c and m between 2 and 40.

shown in Figure 5.2, where `MAX_ALPHABET_SIZE` is c^2 and `LOG2ALPHA` is the number of bits necessary to represent the alphabet. This implementation works only if c^k is a representable integer.

The alphabet cannot be too large, because of the space used. For example, for ASCII code and $k = 2$ we need 16Kb of main memory (for $m < 256$). We have tested the new algorithm with $k = 2$ using a random text of 40000 characters with different alphabet sizes and an English text of approximately 47600 characters using an alphabet of size 32 (lower case letters plus space and punctuation). In both cases the results are the average of 100 runs. The pattern is random for random text, and random words from the text in the case of English text. We have used two measures: the expected number of comparisons between characters and the average

```
k2search( text, n, pat, m )
char text[], pat[];
int n, m;
{
    int i, j, k, mp;
    char d[MAX_ALPHABET_SIZE];

    mp = m-1;
    for( k=0; k<MAX_ALPHABET_SIZE; k++ ) d[k] = mp;
    for( k=1; k<mp; k++ ) d[pat[k] << LOGALPHA + pat[k+1]] = mp-k;
    pat[0] = CHARACTER_NOT_IN_THE_TEXT;
    text[0] = CHARACTER_NOT_IN_THE_PATTERN;
    for( i = m; i <= n; i += d[text[i-1] << LOGALPHA + text[i]] )
    {
        for( j=m, k=i; text[k] == pat[j]; j--, k-- );
        if( j == 0 ) Report_match_at_position( k+1 );
    }
}
```

Figure 5.2: Implementation for the case $k = 2$.

searching time.

Note that the results are independent of the actual number of occurrences, because we are searching for all occurrences, so the text is scanned completely. Figure 5.3 shows experimental results for the expected number of comparisons in random text, for $k = 1$ and $k = 2$ and different alphabet sizes. Figure 5.4 shows the average time (in seconds) to search for 1000 random patterns in random text and two different alphabet sizes (preprocessing included). We include also the Boyer-Moore algorithm as presented in Knuth *et al* [KMP77] to compare it with Horspool's version. Figure 5.5 shows the experimental results for the expected number of comparisons for English text.

Our experimental results agree with the theory. From Figure 5.5 we see that, in practice, the breakeven points for $k = 1$ and $k = 2$ are not too far from the theory: $m = 3$ (instead of 4) for $c = 2$; $m = 6$ (instead of 5) for $c = 8$; and $m = 9$ (instead of 7) for $c = 16$. Therefore, the alphabet transformation is of practical importance only for small alphabets; that is, for $c << m$.

5.4 Reducing the Space

The main objection to the previous transformation is the exponential increase in the space needed. We can reduce the space by using the previous transformation only with the characters compared in the last trial.

If we have a mismatch in the last character we use the original Horspool table. If we have a mismatch in the $(m-j)$-th position ($j > 0$), we use the last $j+1$ characters to address another table of shifts that indicates where in the pattern we have the same sequence of $j+1$ characters. However, because we know that the last j characters matched, we need only address the table with the character in the $(m-j)$-th position.

Let β denote the number of tables that we use; that is, if we have a partial match longer than β, we use the β-th table.

Lemma 5.4.1 *The expected shift for β tables is given by*

$$\overline{S}'(\beta) = \beta(c-1) + 1 + (c-1)\sum_{i=1}^{\beta-1}\left(1-\frac{1}{c^i}\right)^{m-i+1} - c\left(1-\frac{1}{c^\beta}\right)^{m-\beta+1},$$

and the "equivalent" length of the string is

$$m'(\beta) = m - \frac{1}{c-1}\left(1-\frac{1}{c^{\beta-1}}\right).$$

Proof: The expected shift is

$$\overline{S}'(\beta) = \sum_{i=1}^{\beta-1}\frac{1}{c^{i-1}}(1-\frac{1}{c})\overline{S}(m-i+1) + \frac{1}{c^{\beta-1}}\overline{S}(m-\beta+1),$$

where $\overline{S}$ is the expected shift in the BMH algorithm. Computing this formula we obtain the first result. The "equivalent" length is

$$m'(\beta) = \sum_{i=1}^{\beta-1}\frac{1}{c^{i-1}}(1-\frac{1}{c})(m-i+1) + \frac{1}{c^{\beta-1}}(m-\beta+1).$$

Computing the summations we obtain the second result. ■

Hence, the above transformation multiplies the alphabet size by a factor of β (approximately) and decreases the length by a small constant. Therefore, by combining this transformation with the one presented in Section 5.2, we can obtain almost any desired apparent alphabet size.

Again, we are interested in the *optimal* value of β. However, it is not possible to get a closed form to the transcendental equation in β (as in section 5.2) because $\overline{S}'(\beta)$ is not in a complete closed form. But, we can compute the optimal value of β by noting that $\overline{S}'(\beta)$ is a convex function and evaluating it. Surprisingly, in the range 2 to 40, for c and m, we have $\beta_{opt} = k_{opt}$. That is, the optimal value of β is also given by Figure 5.1. However, the value of $\overline{S}'(\beta_{opt})$ is different from $\overline{S}(m - k_{opt} + 1)$ and, as expected, it is not difficult to show that

$$\overline{S}'(\beta_{opt}) \leq \overline{S}(m - k_{opt} + 1) \; ;$$

that is, the shift obtained by the transformation of Section 5.2 is better.

To summarize, using this transformation we can improve the BMH algorithm and use less space. This is useful if c is large. A similar set of tables (or a two dimensional table) may be used to combine the information provided by the occurrence and the match heuristic, in a similar vein to Apostolico and Giancarlo [AG86].

5.5 Hybrid Algorithms

In this section we combine two techniques into one to obtain a hybrid algorithm. The common use of a hybrid algorithm, is an algorithm which, depending on the values of the parameters received, chooses the most appropriate method to solve the given problem. For example, we could use the naive algorithm if we have a pattern of length less than four, otherwise the Boyer-Moore algorithm if we know the alphabet size, or the Knuth-Morris-Pratt algorithm if we do not.

By noting that Horspool's variant is *independent of the order of the comparisons* we can obtain hybrid algorithms between the BMH algorithm and other algorithms. We see this in a particular case: a hybrid of the Knuth-Morris-Pratt and Boyer-Moore-Horspool algorithms. The idea is very simple: comparisons are done from left to right, and after a mismatch we shift the pattern using the maximum between the *next* table of the KMP algorithm and the d table of the BMH algorithm.

The shift in the KMP algorithm is given by $\delta_1 = j - next[j]$ if we have a mismatch in position j of the pattern. The shift for BMH algorithm is given by $\delta_2 = d[text_{k+m-j}]$, where k is the current position in the text. We never have to backtrack in the text (as in the BM algorithm), but we have to look ahead at most $m - 1$ positions (a buffer of size $2m$ suffices). At first glance, we would like to use δ_2 if $\delta_2 > \delta_1$. However, if $next[j] > 1$ we lose all the information gathered by the KMP algorithm. The correct condition is $\delta_2 \geq j - 1$ to ensure that we do not go left in the text. Figure 5.6 shows the C code of this algorithm, which has the following properties.

- It requires $2n$ comparisons in the worst case, improving upon any practical variant of the BM algorithm (due to KMP).
- It requires less than n comparisons on the average, an improvement over the KMP algorithm (due to BMH).
- The preprocessing of the pattern is simpler than the one used in the BM algorithm (due to KMP).

Figure 5.7 shows the experimental results in random text for this algorithm, contrasted with the results for the Boyer-Moore and Boyer-Moore-Horspool algorithms (see Section 4.2.3 for details about the experimental data).

Figure 5.8 shows the timing results while searching English text compared with the KMP, BM, and BMH algorithms. The hybrid algorithm is slightly better than the BMH algorithm, and the KMP algorithm is better only for $m = 2$.

Because the BMH algorithm does not depend on the comparison order, we can combine this technique with any other linear time string matching algorithm to improve its average case behaviour. For example, for a set of strings, we may obtain a hybrid with Aho and Corasick's [AC75] algorithm using the d table for the *shortest string*. This provides us with an algorithm that has a truly linear worst case and good average case, improving upon the solution of Commentz-Walter [CW79].

5.6 Searching in English Text

From Figure 5.5, we see that the results for random and English text with $k = 1$ are very similar. However, the main difference between random text and English text is the following: if we have a partial match, the probability that the next character will match is, in general, greater than $1/c$. For example, suppose that we are searching for *computer* and at some point we have *compu* as a partial match. What is the probability that the next character is a t? The answer depends on how many words in English begin with this prefix, which is obviously greater than $1/c$. This tells us that in English text the value of $\overline{C}_m$ (see Lemma 4.2.3) is a little higher.

How do we model this correlation (context dependency) when we have partial matches? Suppose that we are searching for a pattern of length m. Let $q_1 = p_{equal}$, as in random text, be the probability of matching the first character of the pattern and q_i be the probability of matching the i-th character, given a partial match of length $i-1$. Then, we would expect $q_1 \leq q_2 \leq \cdots \leq q_m \leq 1$. Based on the frequency of characters in the English text that we used, we found that p_{equal} is 0.073 and if we do not include space as a valid character for the first letter of a word, the value of p_{equal} is 0.046. This is the value what we would expect to use for the analysis of the naive algorithm that tries all possible substrings.

Figures 5.9 and 5.10 show the values of q_i as seen by four algorithms: The naive and Knuth-Morris-Pratt algorithms (left to right) and the Boyer-Moore and Boyer-Moore-Horspool algorithms (right to left). The results are plotted for 4 different pattern lengths. Note that for the left to right algorithms p_{equal} is close to 0.046 as expected. This is because only prefixes of words (sentences) are used. Another interesting fact is that the probabilities are about the same for both algorithms (the naive algorithm tries all positions, Knuth-Morris-Pratt does not). For the right to left algorithms, position 1 means a mismatch in position m and so on (reverse order). Here, p_{equal} is close to 0.073, because the prefix of a sentence can finish with a space. Again, the probabilities for both cases are similar.

Based on the empirical evidence, a possible model for this probability distribution is an exponential curve

$$Prob\{p_j = t_i/partial\ match\} = 1 - (1 - p_{equal})e^{-\alpha(j-1)} ,$$

where α is a parameter that controls how fast the curve converges to 1. By using the method of least squares, we found that a good approximation for the left to right algorithms is

$$Prob\{p_j = t_i/partial\ match\} = 1 - 0.955e^{-0.26(j-1)} ,$$

and for the right to left algorithms is

$$Prob\{p_j = t_i/partial\ match\} = 1 - 0.92e^{-0.3(j-1)} .$$

Both curves are also plotted in Figures 5.9 and 5.10. It is interesting to see that the correlation in both directions and for all values of m is very similar. Although these experiments are not conclusive, it is clear that the correlation is very high, and consequently to search in English text is *slightly harder* than to search in random text. In cases where a high correlation exists, we can exploit the non-uniform distribution of the symbols in the alphabet to improve the average time on English text. We use English text as an example in the following. A simple test of text randomness is to compute

$$R = c\sum_{i=1}^{c} f_i^2 \geq 1 ,$$

where c is the alphabet size and f_i is the relative frequency of the i-th symbol of the alphabet in the text. For random text we have $R = 1$. Note that R is the second moment of the distribution multiplied by the alphabet size. Thus, if R is close to one, we consider the text to be almost random.

Based only on the pattern, it is not possible to improve the length of the expected shift in the Boyer-Moore-Horspool version. However, it is possible to reduce the number of comparisons in each trial. Suppose we are searching for the pattern

maximize. It is a good idea to see first if z is in the text rather than to compare e, because e is a character with high frequency in English. Thus, the optimal comparison order is given by the probability of finding each character on the text. We start by comparing the least frequent character, and if that character matches, we continue with the next least frequent, and so on. We can find the frequencies by preprocessing the text, or using known statistics for the given language.

This new algorithm is presented in Figure 5.11, where `Optimal_order` is the function that computes the optimal order for the comparisons using an ad-hoc table of probabilities.

The main drawback of the algorithm is that we need $O(m \log m + c)$ preprocessing time and $O(m+c)$ space. Also, the internal loop is slightly more complex. A simpler version of this heuristic that optimizes only the last two positions of the pattern (where we expect to have differences) improves the BMH algorithm in English text (see Figure 5.12) This improvement is greater if the probability distribution of the symbols is very skewed.

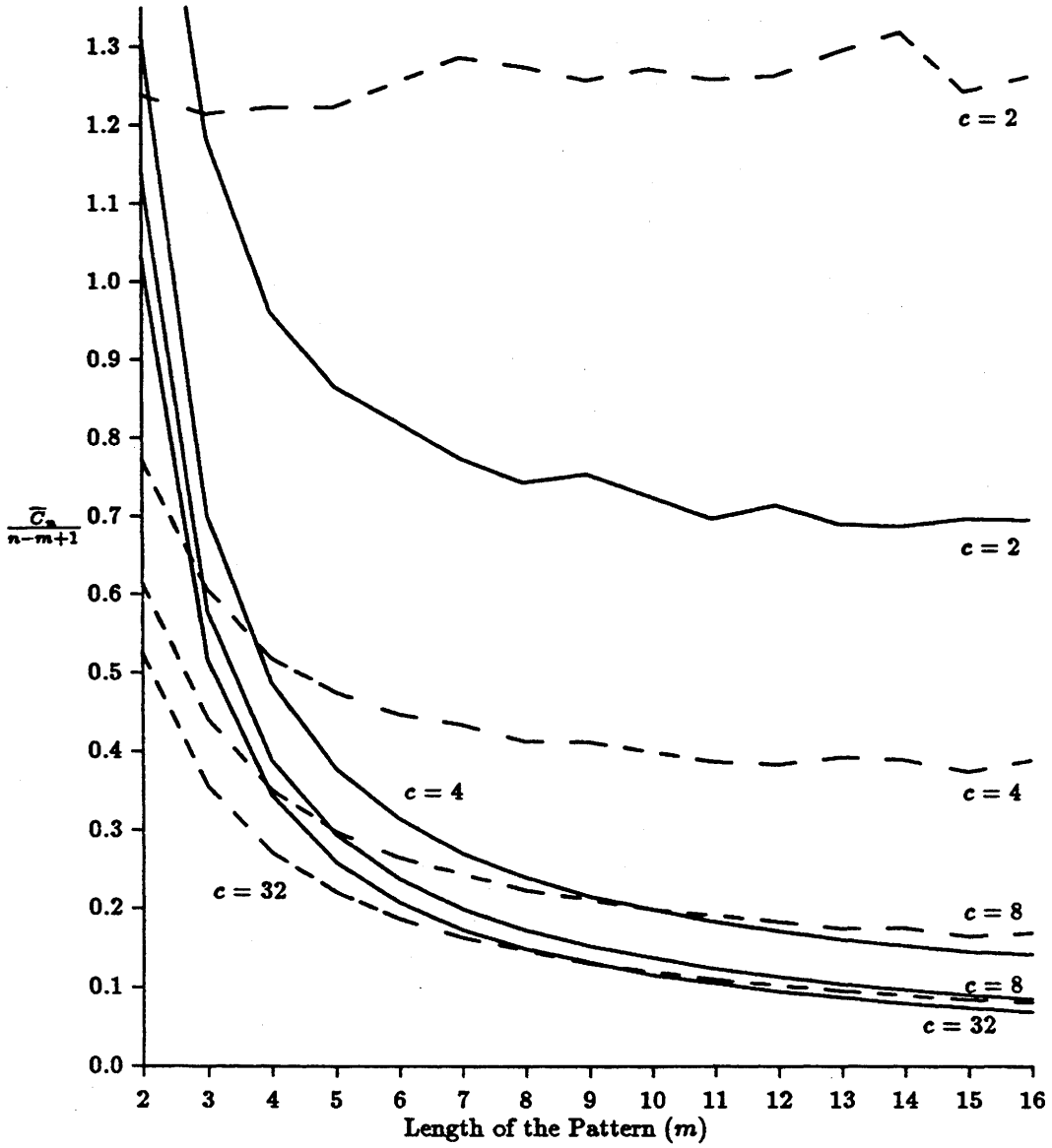

Figure 5.3: Simulation results for the expected number of comparisons in random text for $k = 1$ (dashed line) and $k = 2$ for different alphabet sizes.

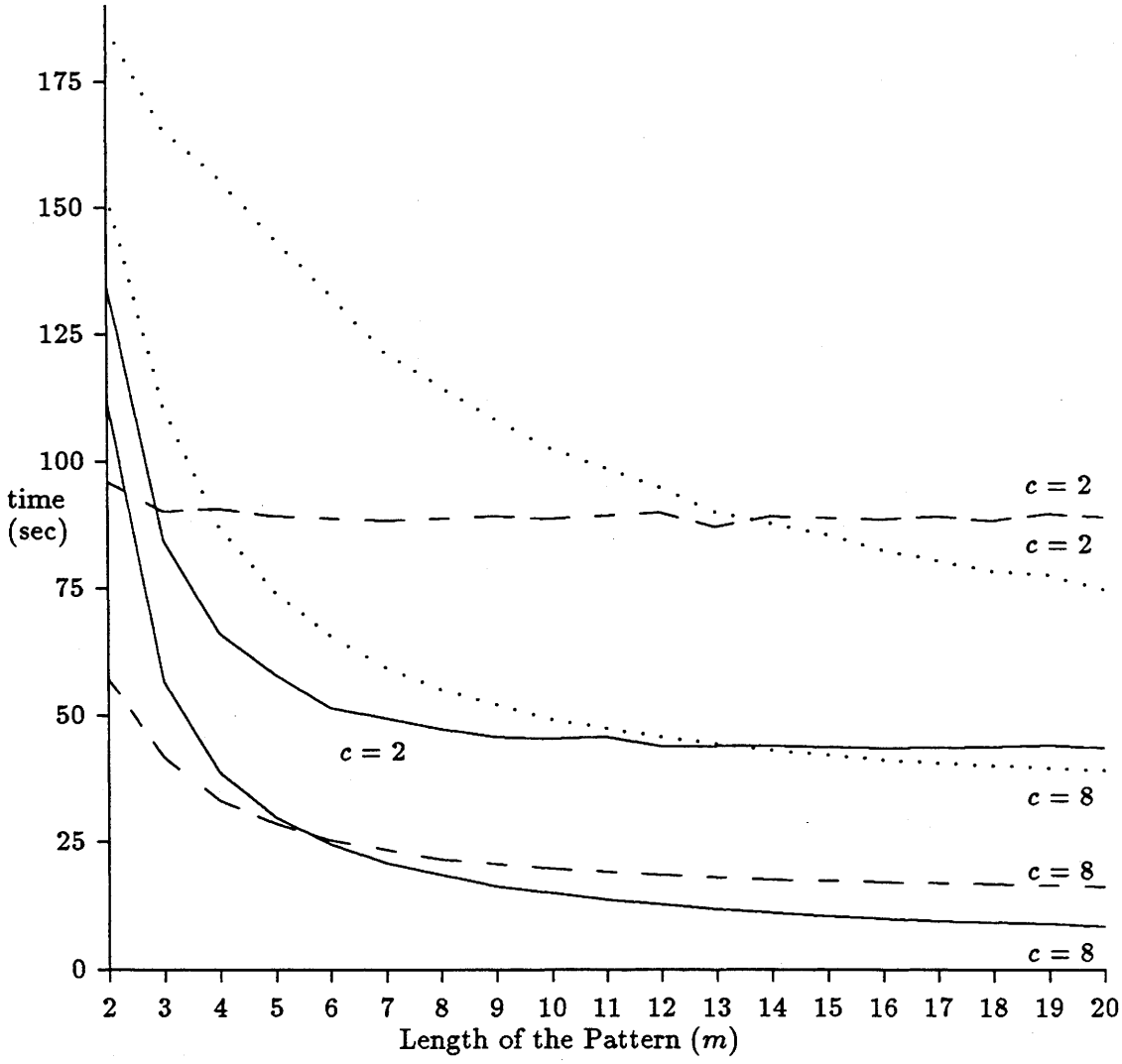

Figure 5.4: Average time to search 1000 random patterns in random text for the Boyer-Moore algorithm (dotted line), $k = 1$ (dashed line) and $k = 2$ (solid line).

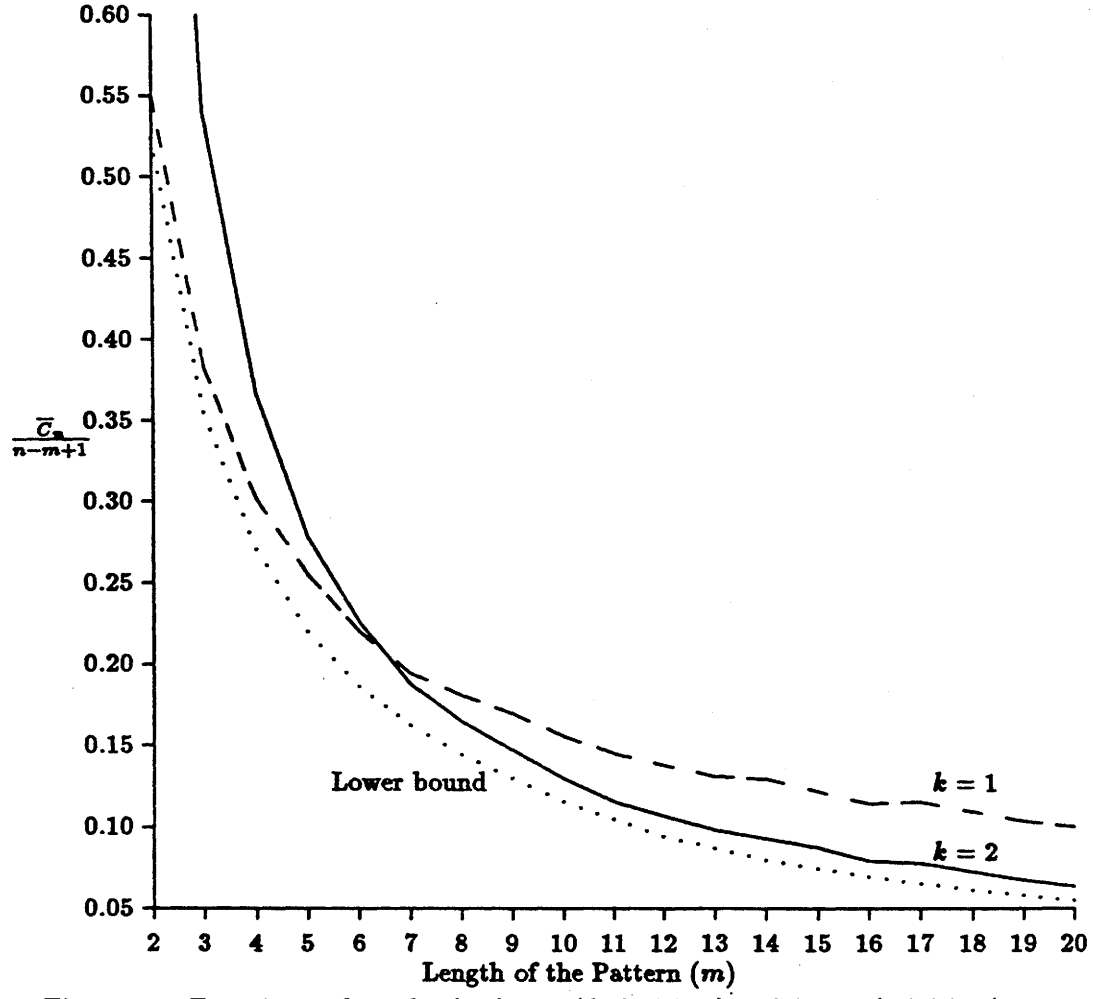

Figure 5.5: Experimental results for $k = 1$ (dashed line) and $k = 2$ (solid line) in English text (the dotted line is the lower bound for optimal k).

```
hybridsearch( text, n, pat, m )
char text[], pat[];
int n, m;
{
    int i, j, lim, m1, shift, resume;
    int d[MAX_ALPHABET_SIZE], first;  /* d table */
    int next[MAX_PATTERN_SIZE+1];     /* next table */

    initd( pat, m, d );          /* Preprocess the pattern */
    pat[m+1] = CHARACTER_NOT_IN_TEXT;
    initnext( pat, m+1, next ); /* next table */
    /* Special values */
    first = pat[1]; resume = next[m+1]; next[m+1] = -1;
    m1 = m - 1; lim = n - m1; i=1;
    /* Search */
    do {
        if( text[i] == first ) /* enter KMP mode */
        {
            shift = d[text[i+m1]] + 1;
            j=2; i++;
            do {
                while( text[i] == pat[j] ) { i++; j++; }
                if( shift >= j )     /* back to BMH mode (j<=m) */
                   {
                    i += shift - j;
                    break;
                }
                j = next[j];         /* next state */
                if( j == 0 ) { i++; j++; }
                else if( j < 0 )
                {
                    Report_match();  /* at position i-m */
                    j = resume;
                }
                /* j >= 1 at this point */
                if( j == 1 || i-j > lim ) break;  /* back to BMH mode */
                shift = d[text[i+m-j]] + 1;       /* update shift */
            } while( 1 );
        }
        else i += d[text[i+m1]];
    } while( i <= lim );
    pat[m+1] = 0;
}
```

Figure 5.6: KMP-BMH Hybrid algorithm.

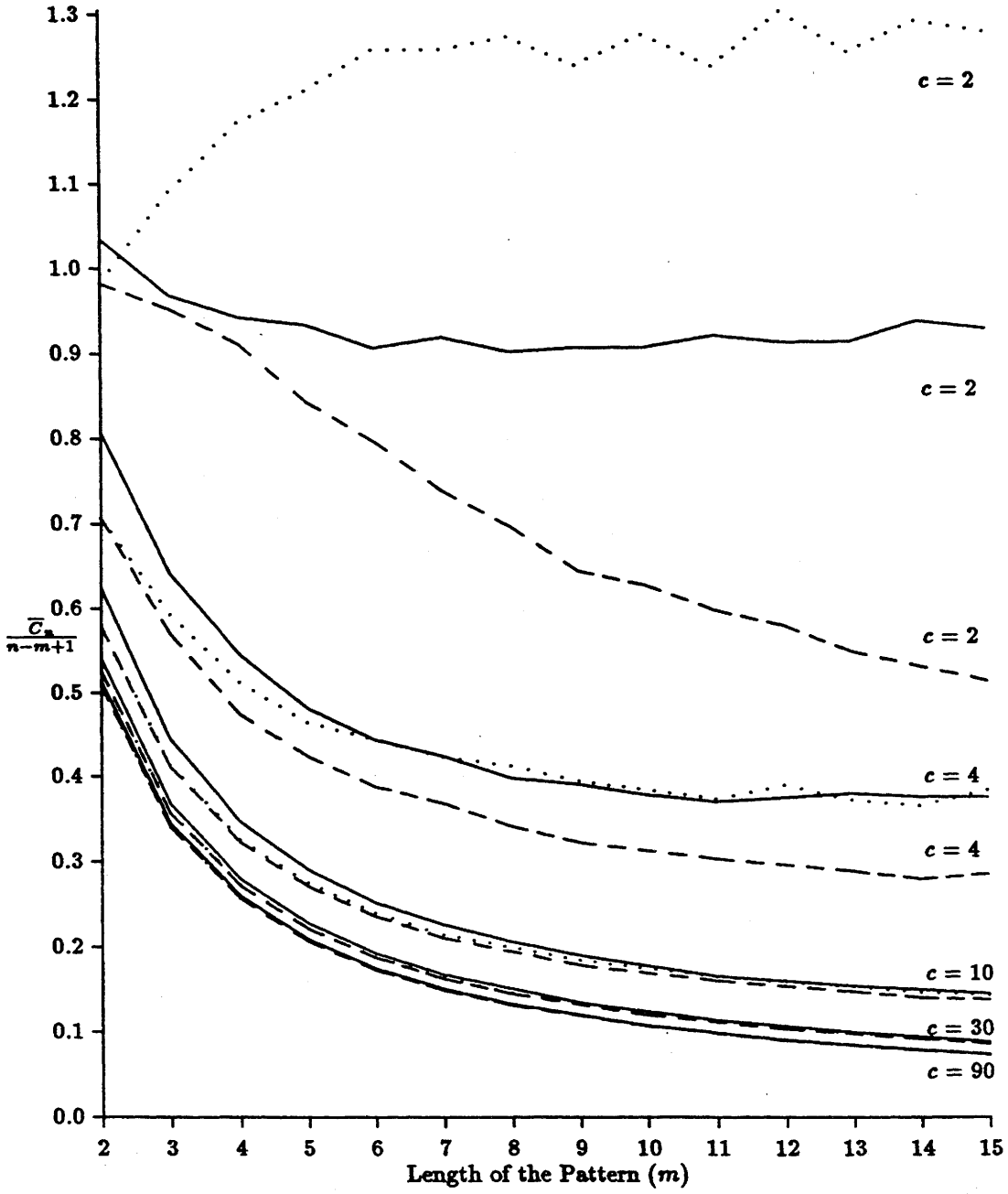

Figure 5.7: Experimental results for the hybrid algorithm.
(The dashed line is the BM algorithm, and the dotted line the BMH algorithm.)

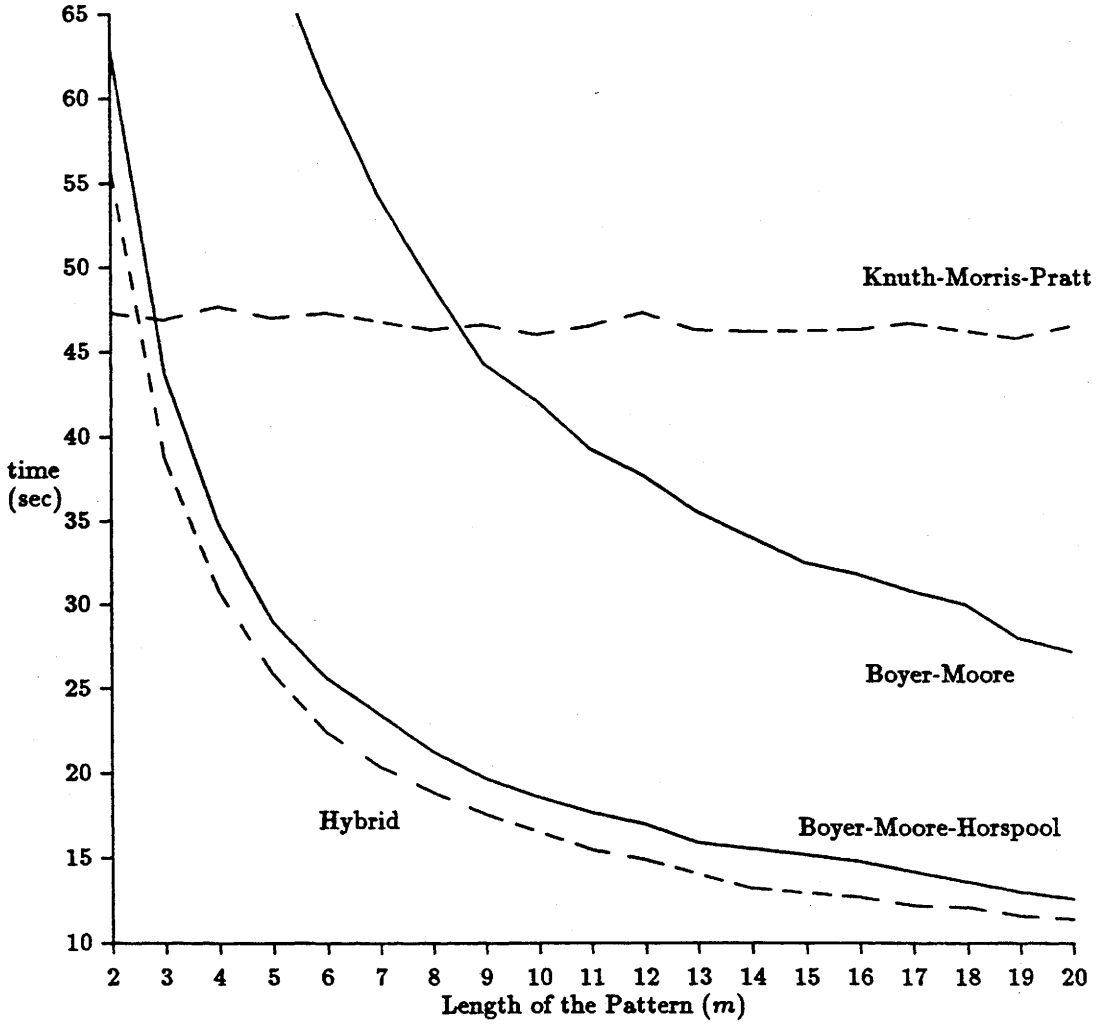

Figure 5.8: Timing results for the hybrid algorithm contrasted with other algorithms while searching for 1000 patterns in English text.

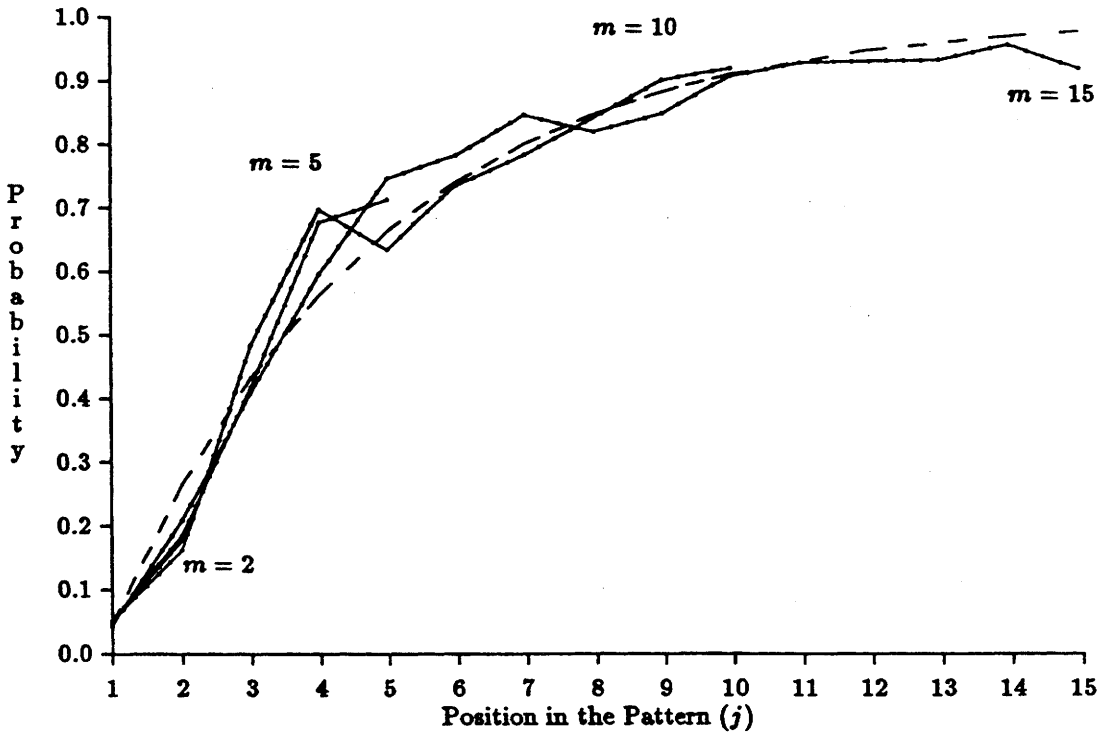

Figure 5.9: Values for q_j in the naive (dotted line) and Knuth-Morris-Pratt algorithms.
(The dashed line represents the approximation given by the model.)

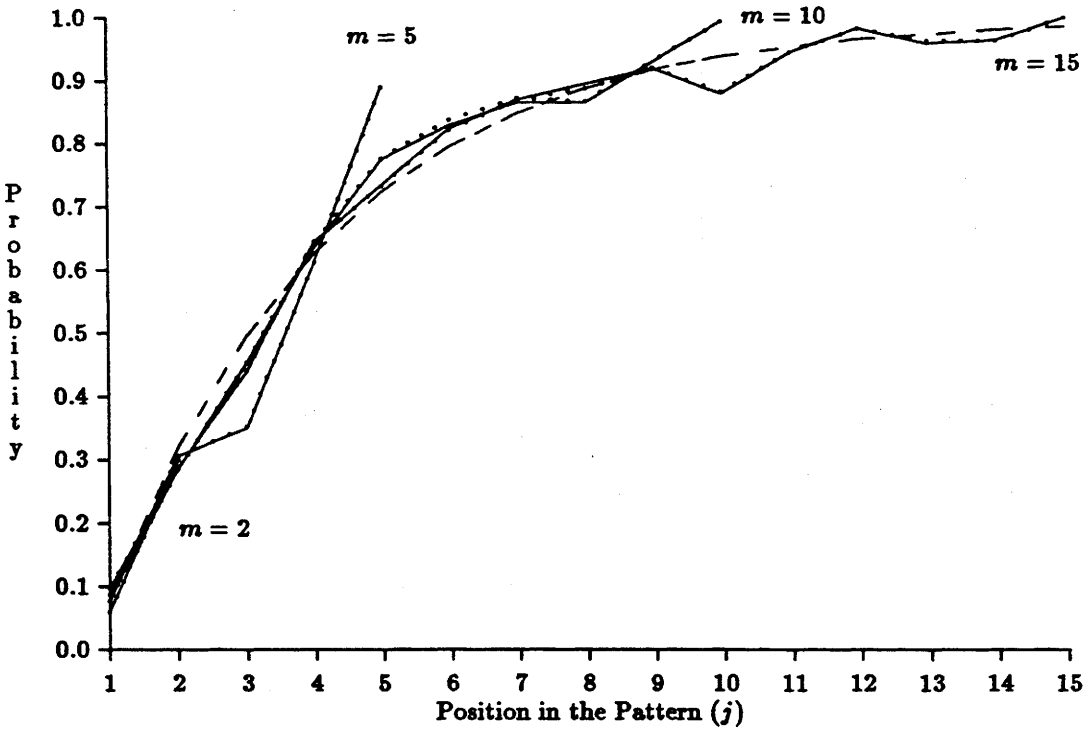

Figure 5.10: Values for q_j in the Boyer-Moore (dotted line) and BMH algorithms. (The dashed line represents the approximation given by the model.)

```
for( k=0; k<MAXSYM; k++ ) d[k] = m;
for( k=1; k<m; k++ ) d[pat[k]] = m-k;
Optimal_order( pat, pos );
k = m;
while( k <= n )
{
    for( j=m; j>0 && text[k-pos[j]] == pat[pos[j]]; j-- );
    if( j == 0 ) Report_match();
    k += d[text[k]];
}
```

Figure 5.11: Boyer-Moore-Horspool algorithm for non-uniform text.

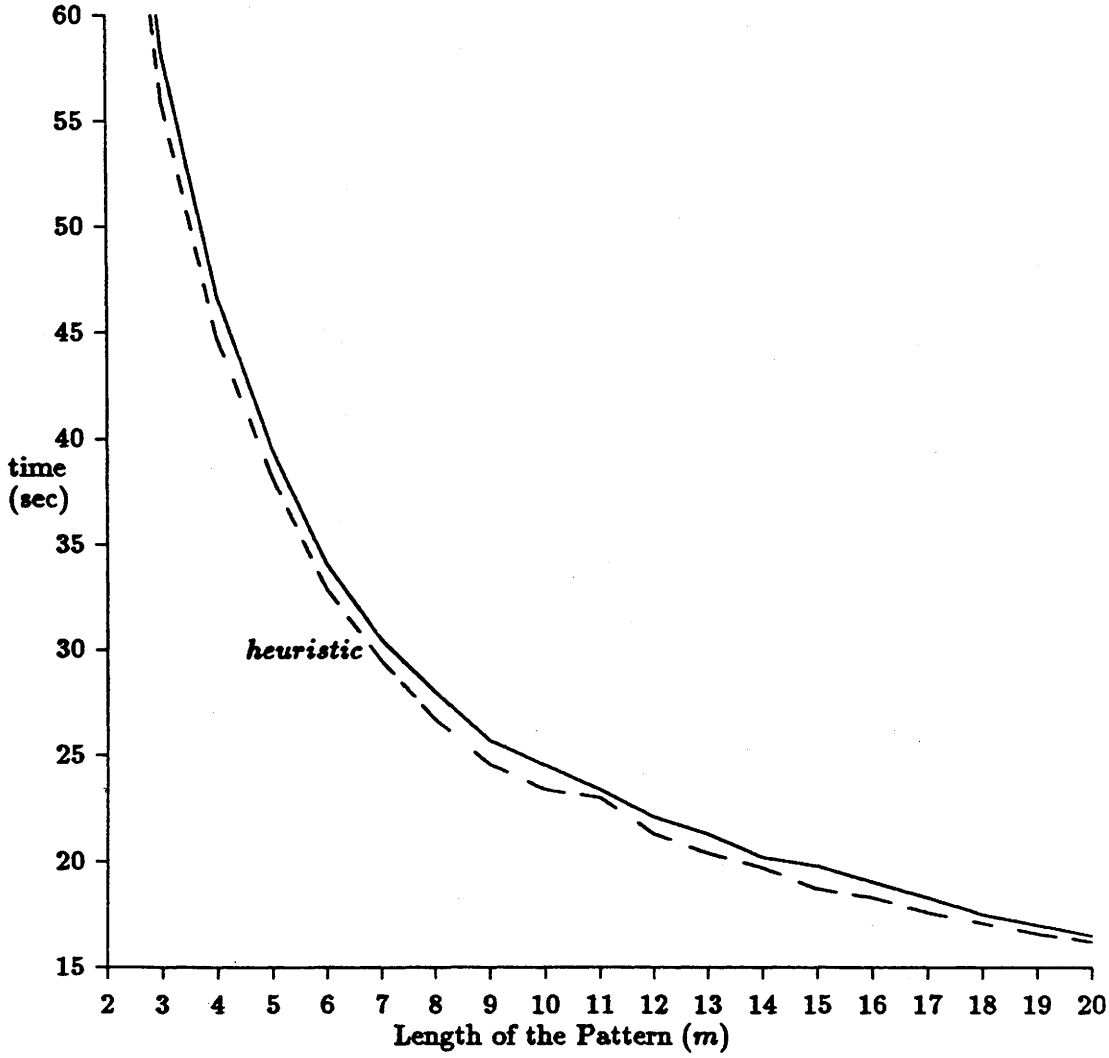

Figure 5.12: Average execution time for searching 1000 patterns in English text for the Boyer-Moore-Horspool algorithm and the heuristic variation (dashed line).

Chapter 6

String Matching with Mismatches

1606 SYLVESTER *Du Bartas* ii. iv. ii. Argt. 6
Mis-Matches taxt.
OED2, **mismatch**

1971 *Nature* 2 4 Dec. 440/3
to the transplanter it may well be that if an antibody
to a particular antigen can be elicited .. certain
typing mismatches may be of no consequence.
OED2, **transplanter**

In this chapter we describe and analyze three simple and fast algorithms for solving the problem of string matching with at most k mismatches. These are the naive algorithm, an algorithm based on a Boyer-Moore approach, and ad-hoc deterministic finite automaton searching algorithm.

6.1 Introduction

The problem of string matching with k mismatches consists of finding all occurrences of a pattern of length m in a text of length n such that in at most k positions the text and the pattern have different symbols. The case of $k = 0$ is the string matching problem. We assume that $0 \leq k < m$ and $m \leq n$, otherwise the problem is trivial.

Landau and Vishkin [LV86] gave the first efficient algorithm to solve this particular problem. Their algorithm uses $O(k(n + m \log m))$ time and $O(k(n + m))$ space.

While it is fast, the space required is unacceptable for most practical purposes. Galil and Giancarlo [GG86] improved this algorithm to $O(kn + m \log m)$ time and $O(m)$ space. This algorithm is practical for small k; however, if $k = O(m)$ it is not.

We present and analyze the naive or brute-force algorithm to solve this problem. While the worst case time is $O(mn)$, the expected time is only $O(kn)$ without using any extra space. We also present a Boyer-Moore approach to the problem [BM77] that has the same complexity, using $O(m - k)$ extra space, but the probability of the worst case is much lower. Finally, we use finite automata theory to solve the problem in time $O(m^{k+2} + n \log m)$ and $O(m^{k+2})$ space. This algorithm is better when k is comparable to m and m is not too large.

6.2 Naive Algorithm

The naive algorithm is basically, for each possible position, count the number of mismatches found. If more than k have been found, try the next position. When we reach the end of the pattern we report a match. Clearly, the worst case number of comparisons is $m(n - m + 1)$. The code in the C programming language for the naive algorithm is presented in Figure 6.1.

```
Search( k, pattern, m, text, n )    /* n >= m, k < m */
int k, m, n;
char pattern[], text[];
{
    int i, j, mismatches;

    for( i=0; i < n-m+1; i++ )
    {
        mismatches = 0;
        for( j=0; j<m && mismatches <= k; j++ )
            if( pattern[j] != text[i+j] ) mismatches++;
        if( j == m )
            Report_match_at_position( i, mismatches );
    }
}
```

Figure 6.1: The naive algorithm for string matching with mismatches.

Let the text and the pattern be random strings of length n and m, respectively, over an alphabet of size $c > 1$. The probability that two symbols, one from the

pattern and one from the text, are equal is $p = 1/c$.

Lemma 6.2.1 *Let $\overline{C}_m^k$ be the average number of comparisons between a random pattern and a random text of lengths m needed to decide if there are at most k mismatches between the strings. Then,*

$$\overline{C}_m^k = \frac{k+1}{1-p} - O(p^{m-k}(1-p)^{k+1}m^k) \ .$$

Proof: We have

$$\overline{C}_m^k = \sum_{j=1}^{m} 1 \times P_{j-1,k} \ ,$$

where $P_{j,k}$ is the probability of at most k mismatches in the first j characters of the text, because we perform one comparison at position j with probability $P_{j-1,k}$.

Clearly $P_{j,k} = 1$, if $j \leq k$; hence, we can define $P_{j,k}$ recursively as

$$P_{j,k} = pP_{j-1,k} + (1-p)P_{j-1,k-1}$$

The solution to this recurrence is

$$P_{k+j,k} = \begin{cases} 1 - (1-p)^{k+1} \sum_{i=0}^{j-1} \binom{k+i}{i} p^i & j > 0 \ . \\ 1 & j \leq 0 \ . \end{cases}$$

Hence, for $m > k$, we have

$$\begin{aligned} \overline{C}_m^k &= k + 1 + \sum_{j=1}^{m-k-1} P_{k+j,k} \\ &= m - (1-p)^{k+1} \sum_{j=1}^{m-k-1} \sum_{i=0}^{j-1} \binom{k+i}{i} p^i \ . \end{aligned}$$

But $\binom{k+i}{i} = (-1)^i \binom{-(k+1)}{i}$, therefore

$$\begin{aligned} \overline{C}_m^k &= m - (1-p)^{k+1} \sum_{j=1}^{m-k-1} \sum_{i=0}^{j-1} \binom{-(k+1)}{i} (-p)^i \\ &= m - (1-p)^{k+1} \sum_{i=0}^{m-k-2} (m-k-1-i) \binom{-(k+1)}{i} (-p)^i \ . \end{aligned}$$

Using the binomial theorem we obtain

$$\begin{aligned}
\overline{C}_m^k &= m - (1-p)^{k+1}\left((m-k-1)(1-p)^{-(k+1)} - p(k+1)(1-p)^{-(k+2)}\right. \\
&\quad \left. - \sum_{i \geq m-k-1} (m-k-1-i)\binom{-(k+1)}{i}(-p)^i\right) \\
&= \frac{k+1}{1-p} + (1-p)^{k+1} \sum_{i \geq m-k-1} (m-k-1-i)\binom{k+i}{i} p^i \\
&= \frac{k+1}{1-p} - O(p^{m-k}(1-p)^{k+1} m^k) \; .
\end{aligned}$$

■

THEOREM **6.2.1** *The average number of comparisons performed by the naive algorithm in a text of length n is*

$$\overline{C}_n^k \leq \frac{c(k+1)}{c-1} n \leq 2(k+1)n \; .$$

Proof: For a text of length n, we have $n - m + 1$ trials; that is,

$$\overline{C}_n^k = \overline{C}_m^k (n - m + 1),$$

and using the previous lemma with $p = 1/c$ and $c > 1$, we get the desired result. ■

For $k = 0$ (brute force string matching), following the same steps of Lemma 6.2.1 and the previous theorem we have

$$\overline{C}_n = \frac{1-p^m}{1-p}(n-m+1) = \frac{c}{c-1}\left(1 - \frac{1}{c^m}\right)(n-m+1) \; ,$$

a result already obtained as Theorem 4.4.1.

Figure 6.2 shows the experimental results (100 trials) for an English text of size approximately 50,000 characters and an alphabet of size 32 (lower case letters plus some other symbols). In the same graph the theoretical results are shown. The agreement is very good, and the differences are because English text is not random (see Section 5.6). For values of k closer to m it may be better to count matches rather than mismatches. That is, string matching with at least $m - k$ matches. In this case

$$\frac{\overline{C}_n^{m-k}}{n-m+1} = (m-k+1)c - O(p^{k+1}(1-p)^{m-k-1} m^{m-k}) \; .$$

Therefore, the breakeven point is $k \approx (m+2)(1-1/c) - 1$, and this is greater than or equal to $m/2$, for $c \geq 2$.

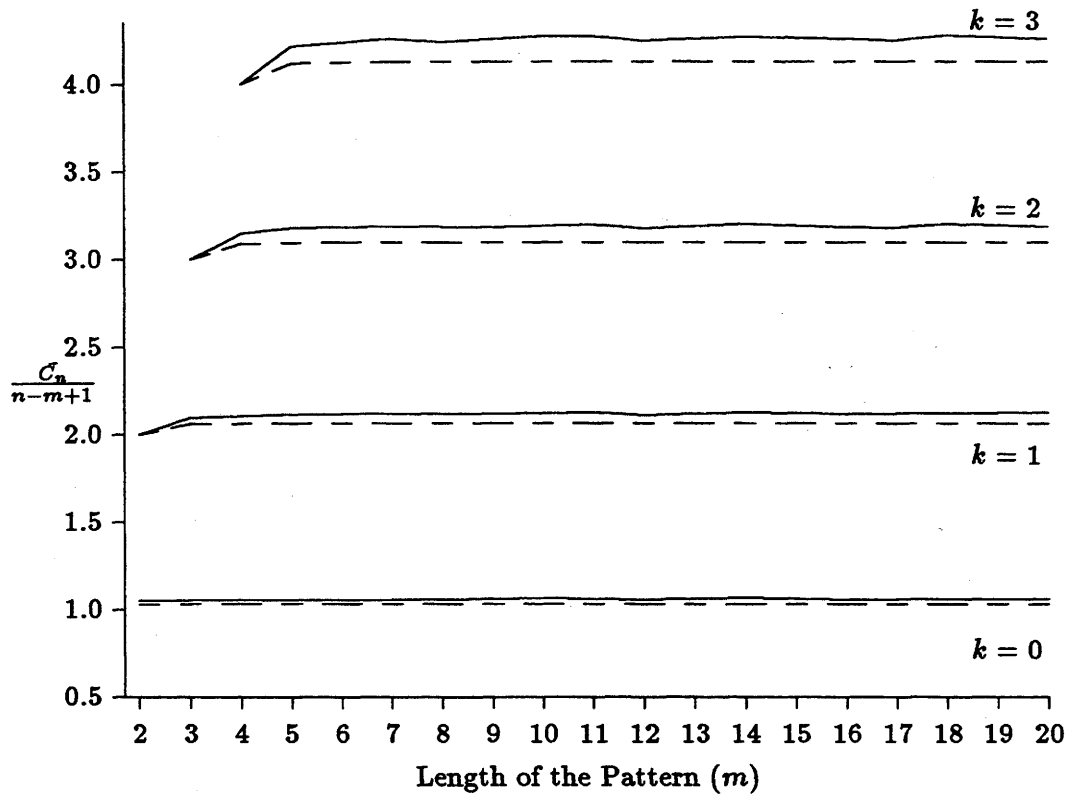

Figure 6.2: Theoretical results for random text (dashed line, $c = 32$) and experimental results for English text with the naive algorithm; for $k = 0, \ldots, 3$.

6.3 A Boyer-Moore Approach

The idea in this method is to search from right to left until we find a match or too many mismatches. At this point, using a precomputed table, we decide how much we can shift the pattern [BM77]. Suppose that we have found a partial match of length j, from the right, with at most k mismatches such that the next character is another mismatch. We define s_j as the maximum shift such that overlapping two copies of the pattern shifted by ℓ characters, for ℓ from 1 to $s_j - 1$, implies that there are at least $2k + 1$ mismatches between both strings. This is to ensure that there are at least $k + 1$ mismatches in the overlap (at most k mismatches are overridden by the mismatches in the partial match).

Clearly s_j, for $j = 0, \ldots, 2k$ is one. To compute the other values of s_j, we slide two copies of the pattern until we find less than $2k + 1$ mismatches. For example, if all the characters of the text are different, then $s_j = m - 2k$, for $j > 2k$. Therefore, this procedure is useful for $k < m/2$. Figure 6.3 shows an example; clearly $s_{j+1} \geq s_j$,

because if we slide s_j positions and we have a partial match of j or more elements, then we have at least $2k+1$ mismatches. Also, because we have more characters in the partial match, potentially we have more mismatches. Using this property, there exists a simple algorithm to set up the table in $m(m-2k)$ worst case time. If all the characters are different only $2mk$ comparisons are necessary. With a slightly more complex algorithm based on Knuth *et al* [KMP77], $O(km)$ preprocessing time can be achieved for any pattern. A simple version for the Boyer-Moore approach is presented in Figure 6.4.

```
pattern:  pointing
           pointing
            pointing
             pointing
              pointing
               pointing
                pointing
  s[j]:   666633111
    j :   876543210
```

Figure 6.3: Example for the table s_j ($k=1$).

The worst case is $2kn$ for many patterns (still $O(mn)$ in general but only for periodic patterns) using $m-2k$ space. On average this algorithm is slightly better than the naive algorithm, improving it for long patterns.

THEOREM **6.3.1** *The average number of comparisons is bounded from below by*

$$\overline{C}_n^k \geq \frac{\overline{C}_m^k}{1+(m-2k-1)P_{2k+1,k}}(n-m+1)\ ,$$

where $P_{2k+1,k}$ is the probability of finding at most k mismatches while comparing $2k+1$ characters, as defined in Lemma 6.2.1.

Proof: In the best case, when all characters are different, the average shift is

$$\overline{S} = 1+(m-2k-1)P_{2k+1,k}\ ,$$

because the shift is $m-2k$ if we compare more than $2k+1$ characters and this happens with probability $P_{2k+1,k}$, otherwise the shift is one. Thus, a lower bound for random text is given by

$$\overline{C}_n^k \geq \frac{\overline{C}_m^k}{\overline{S}}(n-m+1)$$

```
BMmist( k, pattern, m, text, n )    /* n >= m, k < m */
int k, m, n;
char pattern[], text[];             /* m <= MAXPAT */
{
    int i, j, l, mismatches, shift[MAXPAT+1];

    /* Preprocessing */
    for( i=m+1; i>m-2*k; i-- ) shift[i] = 1;
    for( l=1; i>0; i-- )
    {
        for( mismatches=2*k+1; mismatches > 2*k; l++ )
        {
            j = max(1, i-l);
            for( mismatches=0; j<=m-l && mismatches <= 2*k; j++ )
                if( pattern[j] != pattern[j+l] ) mismatches++;
        }
        shift[i] = --l;
    }
    l = n-m+1;
    /* Code to avoid having special cases */
    text[0] = CHARACTER_NOT_IN_THE_PATTERN;
    pattern[0] = CHARACTER_NOT_IN_THE_TEXT;
    /* Search */
    for( i=0; i < l; i += shift[j+2] )
    {
        for( mismatches=0, j=m; j>0 && mismatches <= k; j-- )
            if( pattern[j] != text[i+j] ) mismatches++;
        if( mismatches <= k )
            Report_match_at_position( i+1, mismatches );
    }
}
```

Figure 6.4: Boyer-Moore approach to string matching with mismatches.

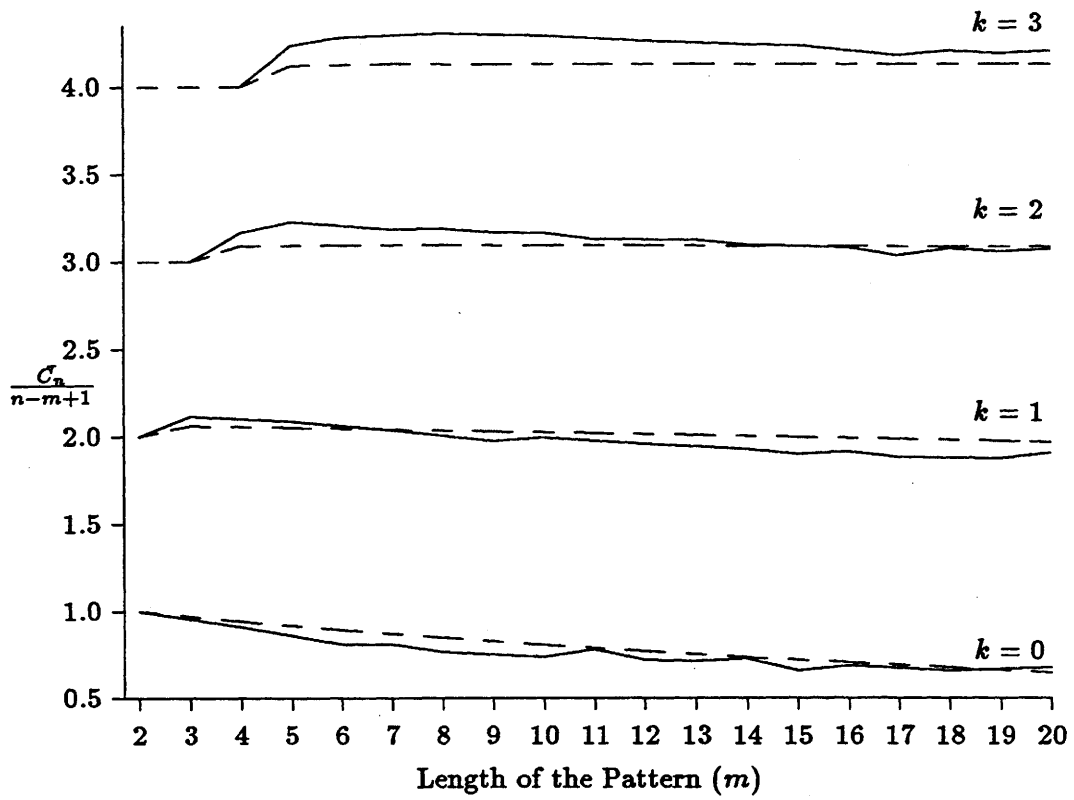

Figure 6.5: Theoretical results for random text (dashed line, $c = 32$) and experimental results in English text (Boyer-Moore approach) for $k = 0, \ldots, 3$.

■

In Figure 6.5 we give the experimental results for English text and the lower bound for random text. For long patterns, this algorithm is clearly better, and we require $(k+1)n + O(mk)$ total expected time and $O(m - 2k)$ space.

6.4 Finite Automaton Approach

The problem can also be stated in terms of regular expressions. For example if we are searching for ab with one mismatch, then that set is described by $\Sigma^*(\Sigma b + a\Sigma)$, where Σ denotes any symbol. In general, there are $\binom{m}{k}$ terms inside the parentheses, and hence the length of the regular expression is $(m+1)\binom{m}{k} + 1$ without counting parentheses. Let r be the regular expression that denotes our searching problem and let $p_m \cdots p_1$ be the pattern. A slightly more compact representation is

$$r = \Sigma^*(S_m^k) ,$$

where

$$S_m^k = \begin{cases} \epsilon & m = 0 \\ p_m S_{m-1}^k & k = 0 \\ \Sigma S_{m-1}^{k-1} & k = m \\ p_m(S_{m-1}^k) + \overline{p_m}(S_{m-1}^{k-1}) & 0 < k < m \end{cases}$$

and where $\overline{x}$ denotes any symbol except x, the complement of x. The number of terms in this case is proportional to $\binom{m+1}{k+1}$.

From this recursive definition, it is very easy to construct a DFA that recognizes the language denoted by r. Figure 6.6 shows the DFA construction for $r = \Sigma^*(a\Sigma + \overline{a}b)$ by using Brzozowski's method of differentiation[Brz64]. Note that we have replaced accepting states (states with ϵ) by output, and we do not have final states and, hence, additional states. Figure 6.7 shows the automaton resulting from this example.

State	*Symbol*	*Regular expression left*	*Nextstate*	*Output*
		r	0	
0	a	$r + \Sigma$	1	
0	$\Sigma - a$	$r + b$	2	
1	a	$r + \Sigma + \epsilon$	1	*match*
1	b	$r + b + \epsilon$	2	*match*
1	$\Sigma - a - b$	$r + b + \epsilon$	2	*match*
2	a	$r + \Sigma$	1	
2	b	$r + b + \epsilon$	2	*match*
2	$\Sigma - a - b$	$r + b$	2	

Figure 6.6: Deterministic finite automaton construction for $r = \Sigma^*(a\Sigma + \overline{a}b)$.

The regular expression after reading a symbol (p_i) (that is, the derivative of r to respect of p_i) is computed by the following formulas

$$r(p_i) = r + S_m^k(p_i)$$

and

$$S_m^k(p_i) = \begin{cases} S_{m-1}^k & p_m = p_i \text{ (if } m = 1, \textit{ match)} \\ S_{m-1}^{k-1} & k = m \text{ or } p_m \neq p_i (m > 0) \\ \emptyset & \text{otherwise} \end{cases}$$

In general we have $O(m^{k+1})$ states and any state can have at most $\min(c, m+1)$ different transitions, where c is the alphabet size. Hence, the size of the automaton

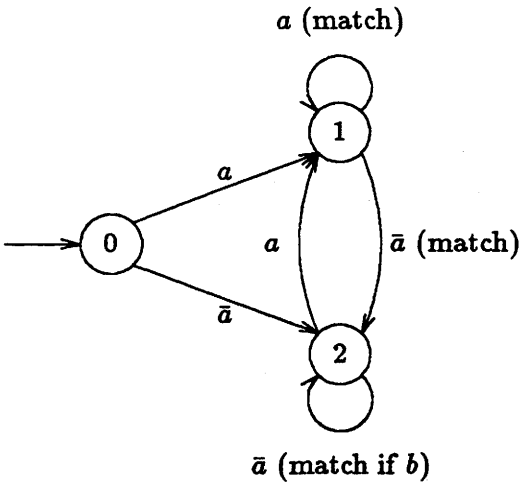

Figure 6.7: Deterministic finite automaton for the pattern ab and $k = 1$.

is $O(m^{k+1} \min(c, m))$. Table 6.1 gives the number of states when all the characters in the string are different for small values of k and m. These observations lead to the following theorem.

THEOREM **6.4.1** *It is possible to construct a DFA to perform string matching with at most k mismatches in $O(m^{k+2})$ space and time. (This reduces to $O(cm^{k+1})$ for an alphabet of size $c < m$.) Using this DFA, searching requires $O(n \log(\min(c, m)))$ time; however, using a complete table for the transitions, we have $O(n)$ search time, but using more space.*

It is worth pointing out that the size of the automaton is, in the worst case, exponential in k and *not* in m.

$m \setminus k$	1	2	3	4
2	3	2		
3	6	7	3	
4	10	16	15	4
5	14	30	43	31

Table 6.1: Number of states for some m and k.

The construction rules are so simple that we can build the automaton as needed during the search. That is, we use $O(m^{k+2})$ worst case time and space only if all possible sequences of mismatches appear in the text. In many applications this is not the case. Another possibility is to describe the automaton as pairs of the form

(m, k), where m indicates the length of the string to be matched and k the maximum number of allowed errors. Clearly, at least k pairs (states) are active at any point in the text and at most m pairs are generated by each symbol in the text. This suggests another $O(kn)$ expected time algorithm with $O(mn)$ worst case time and using $O(m)$ extra space. This approach also suggests that we use a number in base m to represent the pairs, and then if the numbers are smaller than the word size it is possible to have a very efficient algorithm. This idea is pursued in Chapter 7 for string matching with and without mismatches and for more general patterns.

Table 6.2 gives a summary of the complexities of the algorithms presented in this chapter.

Algorithm	Search time		Preprocessing time	Extra space
	Worst case	Average		
LV [LV86]	kn	kn	$km \log m$	$k(n+m)$
GG [GG86]	kn	kn	$m \log m$	m
Naive	mn	kn	1	1
Boyer-Moore approach	mn	kn	$m(m-2k)$	m
(different symbols)	kn	kn	mk	1
DFA	$n \log m$	$n \log m$	m^{k+2}	m^{k+2}
(small alphabet)	n	n	m^{k+1}	m^{k+1}

Table 6.2: Summary of the time and space complexities for string matching with at most k mismatches (order notation).

6.5 Mismatches with Different Costs

All the algorithms presented can be extended to cover the case of different costs for different classes of mismatches. For example, the cost of a mismatch between a vowel and a consonant is twice the cost of a vowel-vowel or consonant-consonant mismatch. In this section we discuss the necessary changes for this case. We assume that there is a finite set of integer costs $\{cost_1, ..., cost_\ell\}$ and we want a match with at most cost C.

For the naive algorithm the solution is trivial. We count the total cost of the mismatches using a multiple if structure for each case. Using more space, we can replace the multiple if by a table indexed by the character in the pattern and the character in the text. This requires $O(|c|^2)$ extra space, where c is the alphabet size.

For the Boyer-Moore approach additionally to the changes in the naive algorithm, we have to modify the table s_j. Now, instead of sliding two copies of the pattern

until we find less than $2k+1$ mismatches, we have to slide it until we have a mismatch with cost less or equal to

$$C + \lfloor C/\min(cost_i) \rfloor \max(cost_i) ,$$

where $\lfloor C/\min(cost_i) \rfloor$ is the maximum number of mismatches in a partial match with at most cost C, and each one of them can override in the worst case a mismatch of maximal cost.

In the case of the DFA, we keep track of the total cost, associating the corresponding mismatch cost to each transition, and we output a match if the total cost is in the appropriate range in the appropriate transitions.

Chapter 7

A New Approach to Pattern Matching

1984 K. BUCKNER *et al. Using UCSD p-System* xv. 156
the WILD unit makes available wild card
pattern matching on string variables.
OED2, **wild**

In this chapter we introduce a family of simple and fast algorithms for solving the classical string matching problem, string matching with "don't care" symbols and complement symbols, and multiple patterns. We also solve the same problems allowing up to k mismatches. Among the features of these algorithms is that they are real time algorithms, that they do not need to buffer the input, and that they are suitable to be implemented in hardware [BYG89b].

7.1 Introduction

The string matching problem consists of finding all occurrences of a pattern of length m in a text of length n. We generalize the problem allowing "don't care" symbols, the complement of a symbol, and any finite class of symbols. We solve this problem for one or more patterns, with or without mismatches. For small patterns the worst case time is linear in the size of the text.

The main idea is to represent the state of the search as a number, and each search step costs a small number of arithmetic and logical operations, provided that the numbers are large enough to represent all possible states of the search. Hence,

for small patterns, we have an $O(n)$ time algorithm using $O(|\Sigma|)$ extra space and $O(m + |\Sigma|)$ preprocessing time, where Σ denotes the alphabet.

For string matching, empirical results show that the new algorithm compares favourably with the Knuth-Morris-Pratt (KMP) algorithm [KMP77] for any pattern length and the Boyer-Moore (BM) algorithm [BM77] for short patterns (up to length 6). For patterns with don't care symbols and complement symbols, this is the first practical and efficient algorithm in the literature, and it can be generalized for any finite class of symbols or their complements.

The main properties of this class of algorithms are:

- Simplicity: the preprocessing and the search are very simple, and only bitwise logical operations, shifts and additions are used.
- Real time: the time delay to process one text character is bounded by a constant.
- No buffering: the text does not need to be stored.

It is worth noting that the KMP algorithm is not a real time algorithm, and the BM algorithm needs to buffer the text.

All these properties indicates that this class of algorithms is suitable for hardware implementation; hence we believe that this new approach is a valuable contribution to all applications dealing with text searching.

7.2 A Numerical Approach to String Matching

Our algorithm is based on finite automata theory, as the Knuth-Morris-Pratt algorithm [KMP77], and also exploits the finiteness of the alphabet, as in the Boyer-Moore algorithm [BM77].

Instead of trying to represent the global state of the search as previous algorithms do, we use a vector of m different states, where state i tell us the state of the search between the positions $1, \ldots, i$ of the pattern and positions $(j - i + 1), \ldots, j$ of the text, where j is the current position in the text.

Suppose that we need b bits to represent each individual state, where as we shall see later, b depends on the searching problem. Then, we can represent the vector state efficiently as a number in base 2^b by:

$$state = \sum_{i=0}^{m-1} s_{i+1} 2^{b \cdot i} ,$$

where the s_i are the individual states. Note that if s_m corresponds to a final state we have to output a match that ends at the current position.

For string matching we need only 1 bit (that is $b = 1$), where s_i is 0 if the last i characters have matched or 1 if not. We have to report a match if s_m is 0, or equivalently if $state < 2^{m-1}$.

To update the state after reading a new character on the text, we must:

- shift the vector state b bits to the left to reflect that we have advanced one position in the text. In practice, this sets the initial state of s_1 to be 0 by default.
- update the individual states according to the new character. For this, we use a table T that is defined by preprocessing the pattern with one entry per alphabet symbol, and an operator *op* that, given the old vector state and the table value, gives the new state. Note that this works only if the operator in the individual state s_i does not produce a carry that affects state s_{i+1}.

Then, each search step is:

$$state = (state \ << \ b) \ \ op \ \ T[curr\ char] \ ,$$

where $<<$ denotes the shift left operation.

The definition of the table T is basically the same for all cases. We define

$$T_x = \sum_{i=0}^{m-1} \delta(pat_{i+1} = x) 2^{b \cdot i} \ ,$$

for every symbol x of the alphabet, where $\delta(C)$ is 0 if the condition C is true, and 1 otherwise. Therefore we need $b \cdot m \cdot |\Sigma|$ bits of extra memory, and if the word size is at least $b \cdot m$, only $|\Sigma|$ words are needed. We set up the table preprocessing the pattern before the search. This can be done in $O(\lceil \frac{mb}{w} \rceil (m + |\Sigma|))$ time.

Example 1: Let $\{a, b, c, d\}$ be the alphabet, and *ababc* the pattern. Then, the entries for the table T are (one digit per position in the pattern):

```
T[a] = 11010      T[b] = 10101      T[c] = 01111      T[d] = 11111
```

A choice for *op* for the case of exact string matching is a bitwise logical or. We finish the example, by searching for the first occurrence of *ababc* in the text *abdababababc*. The initial state is 11111.

```
text :   a     b     d     a     b     a     b     a     b     c
T[x] : 11010 10101 11111 11010 10101 11010 10101 11010 10101 01111
state: 11110 11101 11111 11110 11101 11010 10101 11010 10101 01111
```

For example, the state 10101 means that in the current position we have two partial matches to the left, of lengths two and four respectively. The match at the end of the text is indicated by the value 0 in the leftmost bit of the state of the search. ■

The complexity of the search time in the worst and average case is $O(\lceil \frac{mb}{w} \rceil n)$, where $\lceil \frac{mb}{w} \rceil$ is the time to compute a shift or other simple operation on numbers of mb bits using a word size of w bits. In practice (small patterns, word size 32 or 64 bits) we have $O(n)$ worst and average case time.

For each kind of patterns or searching problem, we can choose b and op appropriately. A similar idea was presented by Gonnet [Gon83] applied to searching text using signatures.

7.3 String Matching with Classes

Now we extend our pattern language to allow don't care symbols, complement symbols and any finite class of symbols. Formally, every position in the pattern can be:

- x: a character from the alphabet.
- Σ: a don't care symbol which matches any symbol.
- [*characters*]: a class of characters, where we allow ranges (for example $a..z$).
- $\overline{C}$: the complement of a character or class of characters C. That is, this matches any character that does not belong to C.

For example, the pattern $[Pp]a\overline{[aeiou]}\Sigma\overline{a}[p..tv..z]$ matches the word *Patter*, but not *python* or *Patton*.

String matching with don't care patterns was addressed before by Fischer and Paterson [FP74] achieving

$$O(n \log^2 m \log\log m \log |\Sigma|)$$

asymptotic search time, and also by Pinter [Pin85] including complement symbols (same complexity). However, these are theoretical results, and their algorithms are not practical. Pinter also gives a $O(mn)$ algorithm that is faster than a naive

algorithm. For small patterns, the complexity of our algorithm is much better, and is much easier to implement. A similar class of patterns is considered by Abrahamson [Abr87], where a theoretical algorithm that runs in

$$O(n + m\sqrt{n}\log n \log\sqrt{\log n})$$

time is presented (in this paper the problem is called "generalized string matching").

Attempts to adapt the KMP algorithm to this case have failed [FP74, Pin85], and for the same reason the BM algorithm as presented in Knuth *et al* [KMP77] cannot solve this problem. It is possible to use the Horspool version of the BM algorithm [Hor80], but the worst case is $O(mn)$; and on average, if we have a don't care character near the end of the pattern, the whole idea of the shift table is worthless. By mapping a class of characters to a unique character, the Karp and Rabin algorithm [KR87] solves this problem too. However, this is a probabilistic algorithm, and if we check each reported match, the search time is $O(n+m+mM)$, where M is the number of matches. Potentially, $M = O(n)$, and their algorithm is slow in practice, because of the use of multiplications.

For this pattern language, we need only to modify the table T, such that, for each position, we process every character in the class. That is

$$T_x = \sum_{i=0}^{m-1} \delta(pat_{i+1} \in Class) 2^{b \cdot i} \ .$$

Example 2: Let $\{a, b, c, d\}$ be the alphabet, and $a\bar{b}[ab]b\overline{[a..c]}$ the pattern. Then, if $b = 1$, the entries for the table T are:

```
T[a] = 11000      T[b] = 10011      T[c] = 11101      T[d] = 01101
```

■

To maintain $O(\lceil \frac{mb}{w} \rceil (m + |\Sigma|))$ preprocessing time (instead of $O(\lceil \frac{mb}{w} \rceil m|\Sigma|)$ time), where m is now the size of the description of the pattern (and not its length), we initially represent T of every symbol in the alphabet as 0 for bit positions corresponding to "don't care" symbols and complements. The search time remains the same.

7.4 Pattern Matching with Mismatches

In this section, we allow up to k characters of the pattern to mismatch with the corresponding text. For example, if $k = 2$, the pattern *mismatch* matches *miscatch* and *dispatch*, but not *respatch*.

Landau and Vishkin [LV86] give the first efficient algorithm to solve this particular problem. Their algorithm uses $O(k(n+m\log m))$ time and $O(k(n+m))$ space. While it is fast, the space required is unacceptable for practical purposes. Galil and Giancarlo [GG86] improve this algorithm to $O(kn+m\log m)$ time and $O(m)$ space. This algorithm is practical for small k. However, if $k = O(m)$, it is not. Other approaches to this problem were presented in Chapter 6.

We solve this problem explicitly only for one pattern, but the solution can be easily extended for multiple patterns (see the next section). In this case one bit is not enough to represent each individual state. Now we have to count matches or mismatches. In both cases, at most $O(\log m)$ bits per individual state are necessary because m is a bound for both, matches and mismatches. Note, too, that if we count matches, we have to complement the meaning of δ in the definition of T. Then, we have a simple algorithm using

$$b = \lceil \log_2(m+1) \rceil$$

and *op* being addition. If $s_m \leq k$ then we have a match. Note that this is independent of the value of k.

Therefore, since $b = O(\log m)$, we need $O(|\Sigma| m \log m)$ bits of extra space. If we assume that we can always represent the value of m in a machine word, we need $O(|\Sigma|m)$ words and preprocessing time. However for small m, we need only $O(|\Sigma|)$ extra space and $O(|\Sigma| + m)$ preprocessing time. For a word size of 32 bits, we can fix $b = 4$ and we can solve the problem for up to $m = 8$.

Clearly only $O(\log k)$ bits are necessary to count the mismatches if we allow at most k mismatches. The problem is that when adding we have a potential carry into the next state. We can get around this problem by having an overflow bit, so that we remember if overflow has happened, but that bit is set to zero at each step of the search. In this case we need

$$b = \lceil \log_2(k+1) \rceil + 1$$

bits. At each step we record the overflow bits in an overflow state, and we reset the overflow bits of all individual states (in fact, we only have to do this each k steps, but such a complication is not practical). Note that if $k > m/2$, then we count matches instead of mismatches (that is, $b = O(\log(m-k))$ if $k > m/2$). The only problem for this case, is that is not possible to tell how many errors there are in a match. Table 7.1 shows maximal values for m when constrained to using a 32 bits word.

Therefore, with a slightly more complex algorithm, we can solve more cases, using only $O(|\Sigma| m \log k)$ extra bits.

Example 3: Let $\{a, b, c, d\}$ be the alphabet, and *ababc* the pattern (see Example 1 for the values of the table T). We want to search for all occurrences of *ababc* with

$k, m-k$	Bits per state	m
0	1	32
1	2	16
2-3	3	10
4	4	8

Table 7.1: Maximum pattern length (m) for a 32 bits word depending on k.

at most 2 mismatches in the text *abdababab c* . Because the value of b is 3 for 2 mismatches, every position in the state is represented by a number in the range 0-4. The initial state is 00000 and the initial overflow is 44444.

```
text :   a     b     d     a     b     a     b     a     b     c
T[x] : 11010 10101 11111 11010 10101 11010 10101 11010 10101 01111
state: 11010 20201 13121 02220 32301 30020 10301 10020 10301 00121
overf: 44440 44400 44000 40000 00000 04000 40000 04000 40000 04000
```

We report a match when the sum of the leftmost digits of the state and the overflow is less than 3. In this case, there is a match at position 4 with one mismatch (detected at position 8), and another match at position 6 with no mismatches (detected at position 10). ■

7.5 Multiple Patterns

We consider briefly, in this section, the problem of string matching with classes for more than one pattern at a time. To denote the union symbol we use "+", for example $p_1 + p_2$ matches the pattern p_1 or the pattern p_2.

The KMP algorithm and the BM algorithm had been extended already to this case (see [AC75] and [CW79] respectively), achieving a worst case time of $O(n+m)$, where m is the total length of the set of patterns.

If we have to search for $p_1 + \cdots + p_s$, and we keep one vector state per pattern, we have an immediate $O(\lceil \frac{mb}{w} \rceil sn)$ time algorithm for a set of s strings. However, we can coalesce all the vectors, keeping all the information in only one vector state and achieving $O(\lceil \frac{mb}{w} \rceil n)$ search time. The disadvantage is that now we need numbers of size $\sum_i |p_i|$ bits, and $O(|\Sigma| \sum_i |p_i|)$ extra space.

In a similar way, we can extend this representation to handle mismatches.

7.6 Implementation

In this section we present efficient implementations of the various algorithms (one per pattern type) that count the number of matches of patterns in a text using one word numbers in the C programming language. Algorithms with different semantic actions in case of a match are easily derived from them.

The programming is independent of the word size as much as possible. We use the following symbolic constants:

- `MAXSYM`: size of the alphabet. For example, 128 for ASCII code.
- `WORD`: word size in bits (32 in our case).
- `B`: number of bits per individual state (1 for string matching).
- `EOS`: end of string (0 in C).

Figure 7.1 shows an efficient implementation of the string matching algorithm. Another implementation is possible using *op* as a bitwise logical *and* operation, and complementing the value of T_x for all $x \in \Sigma$.

Experimental results for searching 100 times for all possible matches of a pattern in a text of length 50K are presented in Table 7.2. For each pattern, a prefix from length 2 to 10 was used (for example, for "representative" the queries were {"re","rep", $\cdots$,"representa"}. The patterns were chosen such that each first letter had a different frequency in English text (from most to least frequent). The timings are in seconds and they have an absolute error bounded by 0.5 seconds. They include the preprocessing time in all cases.

The algorithms implemented are Boyer-Moore, as suggested by Horspool [Hor80] (BMH), which, according to Chapter 4 is the fastest practical version of this algorithm; Knuth-Morris-Pratt, as suggested by their authors [KMP77] (KMP_1), and as given by Sedgewick [Sed83] (KMP_2); and our new algorithm as presented in Figure 7.1 (SO_1), and another version using the KMP_1 idea (SO_2) (that is, do not use the algorithm until we see the first character of the pattern). The changes needed for the latter case (using structured programming!) are shown in Figure 7.2. Note that SO_1 and KMP_2 are independent of the pattern length, that SO_2 and KMP_1 are dependent on the frequency of the first letter of the pattern in the text, and that BMH depends on the pattern length.

From Table 7.2 we can see that SO_2 outperforms KMP_1, being between a 40% and 50% faster. Also it is faster than BMH for patterns of length smaller than 4 to 9, depending on the pattern. Compared with the "grep" program (Berkeley Unix operating system) our algorithm is between 45% and 30% faster (see Table 7.2), in spite of "grep" using faster input routines (low level).

```
Faststrmat( text, pattern )
register char *text;
char *pattern;
{
    register unsigned int state, lim;
    unsigned int T[MAXSYM];
    int i, j, matches;
    if( strlen(pattern) > WORD )
        Error( "Use pattern size <= word size" );
    /* Preprocessing */
    for( i=0; i<MAXSYM; i++ ) T[i] = ~0;
    for( lim=0, j=1; *pattern != EOS; lim |= j, j <<= B, pattern++ )
        T[*pattern] &= ~j;
    lim = ~(lim >> B);
    /* Search */
    matches = 0; state = ~0;              /* Initial state */
    for( ; *text != EOS; text++ )
    {
        state = (state << B) | T[*text]; /* Next state */
        if( state < lim )
            matches++; /* Match at current position-len(pattern)+1 */
    }
    return( matches );
}
```

Figure 7.1: Shift-Or algorithm for string matching (simpler version).

```
    initial = ~0; first = *pattern;
    do {
        do {
            state = (state << B) | T[*text]; /* Next state */
            if( state < lim ) matches++;
            text++;
        } while( state != initial );
        while( *(text-1) != EOS && *text != first ) /* Scan */
            text++;
        state = initial;
    } while( *(text-1) != EOS );
```

Figure 7.2: Shift-Or algorithm for string matching.
(Version based on implementation of [KMP77].)

	Pattern: epresentative						Pattern: representative					
m	BMH	KMP_1	KMP_2	SO_1	SO_2	grep	BMH	KMP_1	KMP_2	SO_1	SO_2	grep
2	36.5	24.4	58.7	30.2	15.8	21.3	23.6	15.5	49.9	30.2	13.2	17.9
3	25.2	24.3	59.0	30.2	15.7	21.1	16.2	15.0	50.2	30.4	13.0	19.8
4	20.5	24.5	58.7	30.2	15.6	21.6	12.6	15.0	50.1	30.2	13.1	20.0
5	17.3	24.3	58.8	30.4	15.8	21.5	11.0	15.2	50.1	30.4	13.1	19.6
6	15.3	24.4	58.7	30.3	15.9	22.0	9.6	15.1	50.9	30.6	13.4	19.3
7	13.2	24.3	58.6	30.1	15.7	21.5	9.0	15.3	50.8	30.5	13.1	19.6
8	12.5	24.4	58.6	30.4	15.6	21.9	7.9	15.3	50.7	30.6	13.3	19.3
9	11.6	24.4	59.7	30.1	15.8	22.0	7.5	15.3	50.5	30.7	13.3	19.4
10	11.2	24.3	58.3	30.1	15.8	21.8	7.1	15.4	50.1	30.2	13.0	19.8
	Pattern: legislative						Pattern: kinematics					
m	BMH	KMP_1	KMP_2	SO_1	SO_2	grep	BMH	KMP_1	KMP_2	SO_1	SO_2	grep
2	37.7	21.0	58.2	30.6	11.9	19.1	35.2	19.0	57.6	30.2	10.4	19.1
3	25.6	21.0	58.6	31.1	12.3	19.3	24.9	19.0	57.4	30.1	10.5	18.1
4	19.9	20.9	57.8	30.4	11.8	20.0	19.9	18.8	57.4	29.9	10.4	18.4
5	16.5	20.6	57.8	30.1	11.7	19.6	16.7	19.0	57.4	30.0	10.4	18.5
6	14.3	20.6	58.0	30.2	11.6	19.2	14.3	19.1	57.6	30.1	10.4	18.6
7	12.9	20.5	57.5	30.1	11.8	19.8	13.0	19.0	57.5	30.1	10.4	18.7
8	12.0	20.6	57.9	30.3	12.0	19.7	12.2	19.0	57.6	30.0	10.4	17.9
9	11.2	20.7	57.7	30.3	12.1	19.7	10.8	19.0	57.3	30.1	10.6	18.2
10	10.3	20.9	58.2	30.3	11.8	19.7	10.0	19.1	57.5	30.2	10.5	18.8

Table 7.2: Experimental results for prefixes of 4 different patterns (time in seconds).

Figure 7.3 shows the preprocessing phase for patterns with classes, using "^" as the complement character and "\" as escape character to represent special symbols (for example, "\^"). The search phase remains as before. The search time for this class of patterns is the same as the search time for a string of the same length.

For pattern matching with at most k mismatches and word size 32 bits, we use $B = 4$ and we count matches instead of mismatches, solving the problem up to $m = 8$, as presented in Figure 7.4.

Figure 7.5 shows the changes needed for the case where we use $O(\log k)$ bits per state. Since the code is similar to the exact string matching case, this algorithm is slightly slower and is independent of k.

For multiple patterns, the preprocessing is very similar to the one in Figure 7.3. The only change in the search phase is the match testing condition:

```
if( (state & mask) != mask ) /* Match? */
```

where mask has a bit with value 1 in the position corresponding to the last state bit of each pattern. Note that this indicates that a pattern *ends* at the current position,

and it is not possible to say where the pattern starts without wasting $O(\lceil \frac{mb}{w} \rceil sM)$ time, where M is the number of matches and s the number of patterns.

```
/* Compute length and process don't care symbols and complements */
for( i=0, j=1, len=0, mask=0; *(pattern+i) != EOS; i++, len++, j <<= B )
{
    if( *(pattern+i) == '^' )       /* Complement */
    {
        i++; mask |= j;
    }
    if( *(pattern+i) == '[' )       /* Class of symbols */
    {
        for( ; *(pattern+i) != ']'; i++ )
            if( *(pattern+i) == '\\' ) i++;   /* Escape symbol */
    }
    else if( *(pattern+i) == '\\' ) i++;      /* Escape symbol */
    else if( *(pattern+i) == '.' ) mask |= j; /* Don't care symbol */
}
if( len > WORD ) Error( "Use B*maxlen <= word size" );
/* Set up T */
for( i=0; i<MAXSYM; i++ ) T[i] = ~mask;
for( j=1, lim=0; *pattern != EOS; lim |= j, j <<= B, pattern++ )
{
    compl = FALSE;
    if( *pattern == '^' )  /* Complement */
    {
        i++; compl = TRUE;
    }
    if( *pattern == '[' )   /* Class of symbols */
        for( pattern++; *pattern != ']'; pattern++ )
        {
            if( *pattern == '\\' ) pattern++;  /* Escape symbol */
            if( compl ) T[*pattern] |= j;
            else        T[*pattern] &= ~j;
            if( strncmp(pattern+1,"..",2) == EQUAL ) /* Range of symbols */
                for( k=*(pattern++)+1; k<=*(++pattern); k++ )
                    if( compl ) T[k] |= j;
                    else        T[k] &= ~j;
        }
    else if( *pattern != '.' )  /* Not a don't care symbol */
    {
        if( *pattern == '\\' ) pattern++;  /* Escape symbol */
        if( compl ) T[*pattern] |= j;
        else        T[*pattern] &= ~j;
    }
}
lim = ~(lim >> B);
```

Figure 7.3: Preprocessing for patterns with classes.

```
Fastmist( k, pattern, text ) /* Pattern matching with k mismatches */
int k;                       /* (B=4, WORD=32, MAXSYM=128, EOS=0) */
char *pattern, *text;
{
    int i, j, m, matches;
    unsigned int T[MAXSYM];
    unsigned int mask, state, lim;
    if( strlen(pattern)*B > WORD )
        Error( "Fastmist only works for pattern size <= WORD/B" );
    /* Preprocessing */
    for( i=0; i<MAXSYM; i++ ) T[i] = 0;
    for( m=0, j=1; *pattern != EOS; m++, pattern++, j <<= B )
        T[*pattern] += j;
    lim = (m-k) << ((m-1)*B);
    if( m*B == WORD ) mask = ~0;
    else              mask = j-1;
    /* Search */
    matches = 0; state = 0;      /* Initial state */
    for( i=1; i<m && *text != EOS; i++, text++ )
         state = (state << B) + T[*text];
    for( ; *text != EOS; text++ )
    {
        state = ((state << B) + T[*text]) & mask;
        if( state >= lim ) /* Match at current position-m+1  */
            matches++;     /* with m-(state>>(m-1)*B) errors */
    }
    return( matches );
}
```

Figure 7.4: Pattern matching with at most k mismatches (simpler version).

```
m = strlen(pattern); type = MISMATCH;  /* count mismatches */
if( 2*k > m ) /* Pattern matching with at least m-k matches */
{
    type = MATCH; k = m-k;  /* count matches */
}
B = clog2(k+1) + 1; /* clog2(n) is the ceiling of log base 2 of n */
if( m > WORD/B ) Error( "Fastmist does not work for this case" );
/* Preprocessing */
lim = k << ((m-1)*B);
for( i=1, ovmask=0; i<=m; i++ ) ovmask = (ovmask << B) | (1 << (B-1));
if( type == MATCH )
    for( i=0; i<MAXSYM; i++ ) T[i] = 0;
else
{
    lim += 1 << ((m-1)*B);
    for( i=0; i<MAXSYM; i++ ) T[i] = ovmask >> (B-1);
}
for( j=1; *pattern != EOS; pattern++, j <<= B )
    if( type == MATCH )
        T[*pattern] += j;
    else
        T[*pattern] &= ~j;
if( m*B == WORD ) mask = ~0;
else              mask = j - 1;
/* Search */
matches = 0; state = 0; overflow = ovmask;  ovmask += mask;  /* Initial state */
for( ; *text != EOS; text++ )
{
    state = ((state << B) + T[*text]);
    overflow = ((overflow << B) | (state & ovmask)) & mask;
    state &= ~ovmask;
    if( type == MATCH )
    {
        if( (state | overflow) >= lim )
            matches++; /* Match with more than m-k errors */
    }
    else if( (state | overflow) < lim )
        matches++; /* Match with (state>>(m-1)*B) errors */
}
```

Figure 7.5: Pattern matching with at most k mismatches.

Chapter 8

Conclusions and Further Research

1927 *Brit. Chess Mag.* XLVII. 61
Yates made a characteristic recovery and added to his growing bag the scalps of two 'grand masters'.
OED2, **grand master**

The main objective of this thesis has been to design and analyze algorithms for fast text searching in practice. The main contributions of the thesis are summarized in this chapter, and some open problems are then presented.

8.1 Summary of Contributions

In Chapter 3 we showed that using a Patricia tree, we can search for many types of queries in logarithmic average time, independently of the size of the answer. We also showed that automaton searching in a trie is sublinear in the size of the text on average for any regular expression, this being the first algorithm to achieve this complexity.

In general, however, the worst case is linear. For some regular expressions and a given algorithm it is possible to construct a text such that the algorithm must be forced to inspect a linear number of characters. The pathological cases are generally with periodic patterns or unusual pieces of text that, in practice, are not used very often. For this reason, we were interested in the average case, which in most practical cases is $O(\sqrt{n})$ or less.

The analysis technique used in Chapter 3 merges matrix theory with generating functions. By using matrix algebra it is possible to simplify the analysis of a set of recurrence equations. For example, consider the following problem:

$$\begin{aligned} f_1(n) &= f_1(n/2) + f_2(n/2) + c_1 \ , \quad f_1(1) = 0 \ , \\ f_2(n) &= \alpha f_2(n/2) + f_1(n/2) + c_2 \ , \quad f_2(1) = 0 \ , \end{aligned}$$

for n a power of 2. Suppose that $\alpha = 0$, then we can reduce the problem to

$$f_1(n) = f_1(n/2) + f_1(n/4) + c_1 + c_2 \ , \quad f_1(2) = f_1(1) = 0 \ .$$

Even this particular case appears to be difficult to solve. However, writing the problem as a matrix recurrence, we obtain

$$\vec{f}(n) = \mathbf{H}\vec{f}(n/2) + \vec{c} \ ,$$

and this recurrence can be solved by simple iteration

$$\vec{f}(n) = \sum_{k=0}^{\log_2 n - 1} \mathbf{H}^k \vec{c} \ .$$

Decomposing $\mathbf{H}$ in its Jordan normal form, we obtain the exact solution for $\vec{f}(n)$. In fact, we hide all the dependencies inside the matrix until the last step, where known summations that depend on the eigenvalues of $\mathbf{H}$ have to be solved. We have already applied this technique to other problems[BYG89c].

In Chapter 4, using a random model for text, we have analyzed the best known algorithms for string matching. The empirical results imply that this model is close enough to English text (searching in English text being slightly harder) to predict the average behaviour of these algorithms. Moreover, the empirical results and the theoretical results coincide even more for typical alphabet sizes ($c \geq 10$). One of the important results is the upper bound on the expected number of comparisons of the KMP algorithm. For Boyer-Moore type algorithms, the most important conclusion is that asymptotically in m, the length of the pattern, the constant factor in the linear term of $\overline{C}_n$ depends only on the alphabet size; that is, it does not converge to zero as m increases [Yao79]. Analogously, asymptotically in c, the size of the alphabet, the constant factor depends only on the pattern size; that is, it does not converge to zero for large c, as is the case of an optimal algorithm [Yao79]. The main drawback of Boyer-Moore type algorithms is the preprocessing time and the space required, which depends on the alphabet size and/or the pattern size. For this reason, if the pattern is small (1 to 3 characters long) it is better to use the naive algorithm. If the alphabet size is large, then Knuth-Morris-Pratt's algorithm is a good choice. In all the other cases, in particular for long texts, Boyer-Moore's

algorithm is better. Finally, the Horspool version of the Boyer-Moore algorithm is the best algorithm, according to the execution time, for almost all pattern lengths.

In Chapter 5 we showed that there is a trade-off between time (on the average) and space in the Boyer-Moore-Horspool algorithm. With the proposed transformations we can improve the original algorithm for small alphabets or long patterns. For non-uniform distributions of characters in text (for example, English), we present different heuristics to improve the search time based on the probability distribution of the symbols used. We also show that we can use Horspool's idea to improve the average case of almost any linear time algorithm. Hence, for small alphabets or long patterns, or for applications dealing with non-uniformly distributed text, we can use the ideas presented in this chapter to improve the search time for large dynamic text databases where the preprocessing time of the pattern is not significant.

In Chapter 6 we presented three simple algorithms for string matching with mismatches. Given the simplicity of the Boyer-Moore approach to string matching with mismatches, this method is a good choice compared to Galil and Giancarlo's algorithm [GG86]. For small alphabets the finite automaton approach is the best choice, mainly because its time bound is independent of the maximum number of mismatches k. All these algorithms can be extended to more general patterns (for example "don't care" symbols, a symbol that represents a class of symbols, etc.).

In Chapter 7 we presented a simple class of algorithms that can be used for string matching and other kind of patterns, with or without mismatches. The time complexity achieved is linear for small patterns, and this is the case in most applications. For longer patterns, we need to implement integer arithmetic of the precision needed, using more than a word per number. Still, if the number of words per number is small, our algorithm is a good practical choice. Using VLSI technology to implement a chip that uses a register of 64 or 128 bits that implements this algorithm for a stream of text, extremely fast search time can be achieved.

This type of algorithm may also be used for other matching problems, for example mismatches with different costs (see Chapter 6) or for patterns of the form (see [Pin85])

$$(set\ of\ patterns)\Sigma^*(set\ of\ patterns)\ .$$

8.2 Open Problems and Future Research

In reference to Chapter 3, finding an algorithm with logarithmic search time for any RE query is still an open problem. Another open problem is to derive a lower bound for searching REs in preprocessed text. Besides this, an interesting question is whether there exist efficient algorithms for the bounded "followed-by" problem (that is, $\Sigma^{\leq k}$).

The complexity of searching for subsequences was reported elsewhere [BY89b]. A related problem, is to search for a sequence of substrings. For example, the query

$$abc\Sigma^* def\Sigma^* ghi\ .$$

Using a Patricia tree [Mor68], where each internal node has an ordered list of all the positions associated to the corresponding prefix, as in Section 3.8.1, we can solve this problem in logarithmic time, using $O(n \log n)$ space on average. It is an open question whether the same time complexity can be achieved using $O(n)$ space.

In relation to string matching algorithms, an exact average case analysis of the KMP algorithm and BM type algorithms is still needed. Exploration of new techniques to analyze these algorithms are currently under investigation.

Finally, the algorithms presented in Chapters 6 and 7 should be extended to more general cases, such as, string matching with errors, where we allow insertion and deletion of characters as well as mismatches. Algorithms for this case can be found in [HD80, Ukk85b, Ukk85a, GG88, OM88, LV88, LV89, GP89].

Another related problem is whether there exist efficient "join" algorithms. That is, given a similarity measure [Joh83], a set of patterns, and a set of texts, find all matches of patterns and texts. One example is the spelling checker problem, in which every word in the text is searched in the dictionary or equivalently every entry in the dictionary is searched in the text. Of particular interest are applications in which the size of the patterns is approximately equal to the size of the texts; hence, naively searching for every pattern in the text gives quadratic time. Are there better algorithms?

Bibliography

[Abr87] K. Abrahamson. Generalized string matching. *SIAM J on Computing*, 16:1039–1051, 1987.

[AC75] A.V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *C.ACM*, 18(6):333–340, June 1975.

[AG86] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM J on Computing*, 15:98–105, 1986.

[Aho80] A.V. Aho. Pattern matching in strings. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 325–347. Academic Press, London, 1980.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[AS87] A. Apostolico and W. Szpankowski. Self-alignments in words and their applications. Technical Report CSD-TR-732, Department of Computer Science, Purdue University, West Lafayette, IN 47907, 1987.

[AS89] A. Apostolico and W. Szpankowski. Alignments in words with applications to the evaluation of suffix trees. Department of Computer Science, Purdue University (unpublished manuscript), 1989.

[Bar81] G. Barth. An alternative for the implementation of Knuth-Morris-Pratt algorithm. *Inf. Proc. Letters*, 13:134–137, 1981.

[Bar84] G. Barth. An analytical comparison of two string searching algorithms. *Inf. Proc. Letters*, 18:249–256, 1984.

[Bat79] D.S. Batory. On searching transposed files. *ACM TODS*, 4:531–544, 1979.

[BCJ+85] T. Benbow, P. Carrington, G. Johannesen, F.W. Tompa, and E. Weiner. Report on the New Oxford English Dictionary: User survey. Technical

Report OED-87-05, UW Centre for the New Oxford English Dictionary, Univ. of Waterloo, Ontario, Canada., 1985. (to appear in Int. Journal of Lexicography).

[BGT88] D.L. Berg, G.H. Gonnet, and F.W. Tompa. The New Oxford English Dictionary Project at the University of Waterloo. Technical Report OED-88-01, UW Centre for the New Oxford English Dictionary, 1988.

[BM77] R. Boyer and S. Moore. A fast string searching algorithm. *C.ACM*, 20:762–772, 1977.

[Brz64] J. Brzozowski. Derivatives of regular expressions. *J.ACM*, 11:481–494, 1964.

[BY89a] R. Baeza-Yates. Improved string searching. *Software-Practice and Experience*, 19(3):257–271, 1989.

[BY89b] R. Baeza-Yates. The subsequence graph of a text. In *CAAP'89, XVI Colloquium on Trees, Algebra, and Programming, Lecture Notes in Computer Science 351*, pages 104–118, Barcelona, Spain, March 1989. Also as Tech. Report CS-88-34, Department of Computer Science, University of Waterloo.

[BY89c] R.A. Baeza-Yates. String searching algorithms revisited. In *Workshop in Algorithms and Data Structures*, Ottawa, Canada, August 1989.

[BYG89a] R. Baeza-Yates and G.H. Gonnet. Efficient text searching of regular expressions. In *ICALP'89*, Stresa, Italy, July 1989.

[BYG89b] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. In *Proc. of 12th SIGIR*, Cambridge, Mass., June 1989.

[BYG89c] R.A. Baeza-Yates and G.H. Gonnet. Solving matrix recurrences with applications. Department of Computer Science, University of Waterloo (submitted for publication), May 1989.

[BYR88] R. Baeza-Yates and M. Regnier. Analysis of Boyer-Moore type string searching algorithms. (document in preparation), 1988.

[CGG+88] B. Char, G. Geddes, G. Gonnet, M. Monagan, and S. Watt. *MAPLE Reference Manual, 5th Edition.* WATCOM Publications Limited, 1988.

[Cla82] A. Clausing. Kantorovich-type inequalities. *The American Mathematical Monthly*, 89:314–330, 1982.

[CM65] D. Cox and H. Miller. *The Theory of Stochastic Processes.* Chapman and Hall, London, 1965.

[CP88] M. Crochemore and D. Perrin. Pattern matching in strings. In Di Gesu, editor, *4th Conference on Image Analysis and Processing*, pages 67–79. Springer Verlag, 1988.

[CP89] M. Crochemore and D. Perrin. Two way pattern matching. Technical Report 98-8, L.I.T.P., Univ. Paris 7, Feb 1989. (submitted for publication).

[Cro88] M. Crochemore. String matching with constraints. In *Mathematical Foundations of Computer Science*, Carlsbad, Czechoslovakia, Aug/Sept 1988. Lecture Notes in Computer Science 324.

[CW79] B. Commentz-Walter. A string matching algorithm fast on the average. In *ICALP*, volume 6 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 1979.

[DB86] G. Davies and S. Bowsher. Algorithms for pattern matching. *Software - Practice and Experience*, 16:575–601, 1986.

[Dev82] L. Devroye. A note on the average depth of tries. *Computing*, 28:367–371, 1982.

[dlB59] R. de la Briandais. File searching using variable length keys. In *AFIPS Western JCC*, pages 295–298, San Francisco, CA, 1959.

[Faw89] H. Fawcett. *A Text Searching System: PAT 3.3, User's Guide.* Centre for the New Oxford English Dictionary, University of Waterloo, 1989.

[FP74] M. Fischer and M. Paterson. String matching and other products. In R. Karp, editor, *Complexity of Computation (SIAM-AMS Proceedings 7)*, volume 7, pages 113–125. American Mathematical Society, Providence, RI, 1974.

[FP86] P. Flajolet and C. Puech. Tree structures for partial match retrieval. *J.ACM*, 33:371–407, 1986.

[Fre60] E. Fredkin. Trie memory. *C.ACM*, 3:490–499, 1960.

[Gal79] Z. Galil. On improving the worst case running time of the Boyer-Moore string matching algorithm. *C.ACM*, 22:505–508, 1979.

[Gal81] Z. Galil. String matching in real time. *J.ACM*, 28:134–149, 1981.

[Gal85] Z. Galil. Open problems in stringology. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 1–8. Springer-Verlag, 1985.

[Gan59] F.R. Gantmacher. *The Theory of Matrices (2 Vols).* Chelsea Publishing Company, New York, 1959.

[GG86] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17:52–54, 1986.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

[GO80] L. Guibas and A. Odlyzko. A new proof of the linearity of the Boyer-Moore string searching algorithm. *SIAM J on Computing*, 9:672–682, 1980.

[Gon83] G.H. Gonnet. Unstructured data bases or very efficient text searching. In *ACM PODS*, volume 2, pages 117–124, Atlanta, GA, Mar 1983.

[Gon84] G.H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison-Wesley, London, 1984.

[Gon87] G.H. Gonnet. PAT 3.1: An efficient text searching system, User's manual. UW Centre for the New OED, University of Waterloo, 1987.

[Gon88a] G.H. Gonnet. Private communication, 1988.

[Gon88b] G.H. Gonnet. Efficient searching of text and pictures (extended abstract). Technical Report OED-88-02, Centre for the New OED., University of Waterloo, 1988.

[GP89] Z. Galil and K. Park. An improved algorithm for approximate string matching. In *ICALP'89*, Stressa, Italy, 1989.

[GS80] Z. Galil and J. Seiferas. Saving space in fast string-matching. *SIAM J on Computing*, 9:417–438, 1980.

[GS83] Z. Galil and J. Seiferas. Time-space-optimal string matching. *JCSS*, 26:280–294, 1983.

[HD80] P.A.V. Hall and G.R. Dowling. Approximate string matching. *ACM C. Surveys*, 12:381–402, 1980.

[Hor80] N. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, 10:501–506, 1980.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory.* Addison-Wesley, Reading, Mass., 1979.

[Joh83] J.H. Johnson. *Formal Models for String Similarity.* PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1983.

[Kaz86] R. Kazman. Structuring the text of the Oxford English Dictionary through finite state transduction. Technical Report CS-86-20, Dept. of Computer Science, Univ. of Waterloo, 1986.

[Ker82] B. Kernighan. PIC - a graphics language for typesetting, User manual. In *Unix Manual.* 1982.

[KMP77] D.E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J on Computing*, 6:323–350, 1977.

[Knu73] D.E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Mass., 1973.

[Knu84] D.E. Knuth. *The TEXbook.* Addison-Wesley, Reading, Mass., 1984.

[KR78] B. Kernighan and D. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KR87] R. Karp and M. Rabin. Efficient randomized pattern-matching algorithms. *IBM J Res. Development*, 31:249–260, 1987.

[Lam86] L. Lamport. *LATEX: A Document Preparation System.* Addison-Wesley, Reading, Mass., 1986.

[LM85] P.A. Larson and M. Mota. GTS - tutorial. Dept. of Computer Science, University of Waterloo, 1985.

[LV86] G. Landau and U. Vishkin. Efficient string matching with k mismatches. *Theoretical Computer Science*, 43:239–249, 1986.

[LV88] G. Landau and U. Vishkin. Fast string matching with k differences. *JCSS*, 37:63–78, 1988.

[LV89] G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10, 1989. (to appear).

[Mor68] D. Morrison. PATRICIA-Practical algorithm to retrieve information coded in alphanumeric. *JACM*, 15:514–534, 1968.

[OM88] O. Owolabi and D. McGregor. Fast approximate string matching. *Software - Practice and Experience*, 18:387–393, 1988.

[OS60] A.M. Ostrowski and H. Schneider. Bounds for the maximal characteristic root of a non-negative irreducible matrix. *Duke Math J.*, 27:547–553, 1960.

[Pin85] R. Pinter. Efficient string matching with don't-care patterns. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 11–29. Springer-Verlag, 1985.

[Reg81] M. Regnier. On the average height of trees in digital search and dynamic hashing. *Inf. Proc. Letters*, 13:64–66, 1981.

[Reg88] M. Regnier. Knuth-Morris-Pratt algorithm: An analysis. INRIA, Rocquencourt, France (unpublished), 1988.

[Riv77] R. Rivest. On the worst-case behavior of string-searching algorithms. *SIAM J on Computing*, 6:669–674, 1977.

[Ryt80] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string-searching. *SIAM J on Computing*, 9:509–512, 1980.

[Sch88] R. Schaback. On the expected sublinearity of the Boyer-Moore algorithm. *SIAM J on Computing*, 17:548–658, 1988.

[Sed83] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, Mass., 1983.

[Smi82] G.V. Smit. A comparison of three string matching algorithms. *Software - Practice and Experience*, 12:57–66, 1982.

[Tak86] T. Takaoka. An on-line pattern matching algorithm. *Inf. Proc. Letters*, 22:329–330, 1986.

[Tho68] K. Thompson. Regular expression search algorithm. *C.ACM*, 11:419–422, 1968.

[Ukk85a] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[Ukk85b] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[Yao79] A.C. Yao. The complexity of pattern matching for a random string. *SIAM J on Computing*, 8:368–387, 1979.

Index