

# dawson france

Rue de la Prairie

91146 Villebon/Yvette Cedex France

agence internationale  
d'abonnement

société anonyme au capital de 2 175 000 francs  
r.c. Corbeil 709.801.872 b

INTERNATIONAL SUBSCRIPTION

AGENCY

☎ (1) 69-09-01-22

Télex 600394 F

Télécopieur : (1) 64.54.83.26



27/05/88

PAYMENT OF INVOICE

NO 2616

PAGE: 1

SCE LIBRAIRIE  
BOOK DPT

005444  
WATERLOO (UNIVERSITY OF)  
SUSAN DE ANGELIS-FAC.MATHEMATICS  
DPT COMPUTER SCIENCE  
WATERLOO ONTARIO N2L 3G1  
CANADA

NOTRE COMPTE NO : \*X  
OUR ACCOUNT NO :

SS REF

11 MAY 88

TOTAL:

4.00 \$CA/

4.00 \$CA/

*received  
payment  
cheque # 15869  
Bank of Mtl  
Quebec*

/sd

Encl.

**University of Waterloo  
Department of Computer Science  
Waterloo, Ontario N2L 3G1**

May 11, 1988

**INVOICE**

Dawson France  
Books Dept.  
B.P. 40  
91121 Palaiseau Cedex  
FRANCE

**REPORT(S) ORDERED**

CS-87-66, 87-68

**TOTAL COST**

\$4.00

Would you please make your cheque or bank money order payable to the Computer Science Department, University of Waterloo and forward to my attention.

Thanking you in advance.

Yours truly,



Susan DeAngelis (Mrs.)  
Research Report Secretary  
Computer Science Dept.

/sd

Encl.

001 ON THE AVERAGE CASE OF STRING  
MATCHING ALGORITHM NO CS 87 66

BAEZA-YATES R.

1987

SURFACE

WATERLOO (UNIVERSITY OF)  
SUSAN DE ANGELIS-FAC. MATHEMATICS  
DPT COMPUTER SCIENCE  
WATERLOO ONTARIO N2L 3G1  
CANADA

0735515/00


11 04 88

001 SEARCHING WITH UNCERTAINTY  
NO CS 87 68

BAEZA-YATES R.-CULBERSON J  
1987

SURFACE

WATERLOO (UNIVERSITY OF)  
SUSAN DE ANGELIS-FAC. MATHEMATICS  
DPT COMPUTER SCIENCE  
WATERLOO ONTARIO N2L 3G1  
CANADA

(Règlement art. 116 § 1)	
	<b>DOUANE</b> <b>C1</b>
<i>Peut être ouvert d'office</i>	
N° 284	
PROCÉDURE D'ABONNEMENT	
N° I - 20 du répertoire	
<b>NE PAS TAXER</b>	

# Printing Requisition / Graphic Services

4180

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material.
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

## TITLE OR DESCRIPTION

US-27-66 Beaver Report

DATE REQUISITIONED

Dec 7 1987

DATE REQUIRED

ASAP

ACCOUNT NO.

(Garnet)

REQUISITIONER - PRINT

PHONE

2192

SIGNING AUTHORITY

BLDG. & ROOM NO.

MC 6081E

MAILING INFO -

NAME

DEPT.

S. DeAngelis

US

DELIVER

PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES **32** NUMBER OF COPIES **100**

TYPE OF PAPER STOCK

☐ BOND ☐ NCR ☐ PE ☐ COVER ☐ BRISTOL ☐ **Alpac Ivory** ☐ **140N**

PAPER SIZE

☐ 8 1/2 x 11 ☐ 8 1/2 x 14 ☐ 11 x 17 ☐ **10x14 Glosscoat** ☐ **10 pt Rolland Tint**

PAPER COLOUR

☐ WHITE ☒ **Black** ☐ **Black** ☐ **Black**

PRINTING

☐ 1 SIDE ☒ 2 SIDES ☐ PGS. ☐ PGS. FROM TO

BINDING/FINISHING

☒ COLLATING ☐ STAPLING ☐ HOLE PUNCHED ☐ PLASTIC RING

FOLDING/PADDING

**7x10 saddle stitched**

Special Instructions

**Beaver Cover**

**Both cover and inside in black ink please**

COPY CENTRE

OPER. NO. BLDG. NO. MACH. NO.

DESIGN & PASTE-UP

OPER. NO. TIME LABOUR CODE  
D 0 1  
D 0 1  
D 0 1

TYPESETTING

QUANTITY

P A P 0 0 0 0 0 0 T 0 1  
P A P 0 0 0 0 0 0 T 0 1  
P A P 0 0 0 0 0 0 T 0 1

PROOF

P R F  
P R F  
P R F

NEGATIVES

QUANTITY

OPER. NO.

TIME

LABOUR CODE

F L M C 0 1  
F L M C 0 1  
F L M C 0 1  
F L M C 0 1  
F L M C 0 1

PMT

P M T C 0 1  
P M T C 0 1  
P M T C 0 1

PLATES

P L T P 0 1  
P L T P 0 1  
P L T P 0 1

STOCK

0 0 1  
0 0 1  
0 0 1  
0 0 1

BINDERY

R N G B 0 1  
R N G B 0 1  
R N G B 0 1  
M I S 0 0 0 0 0 B 0 1

OUTSIDE SERVICES

\$ COST

UNIVERSITY OF WATERLOO  
UNIVERSITY OF WATERLOO  
UNIVERSITY OF WATERLOO

COMPUTER SCIENCE DEPARTMENT  
COMPUTER SCIENCE DEPARTMENT  
COMPUTER SCIENCE DEPARTMENT



*On The Average Case  
of String Matching Algorithms*

*Ricardo A. Baeza-Yates*

*Research Report  
CS-87-66*

*December, 1987*

# On The Average Case of String Matching Algorithms

Ricardo A. Baeza-Yates \*

Data Structuring Group  
Department of Computer Science  
University of Waterloo  
Waterloo, Ontario N2L 3G1, Canada

## Abstract

We study the average case of several string matching algorithms. We obtain analytical results in random text. We derive a simple 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 all the algorithms support the analysis. For patterns of length 2, an exact analysis is provide for almost all the known practical algorithms and optimal algorithms. Finally, we model real text, and some improvements to the Boyer-Moore algorithm are presented.

## 1 Introduction

This work deals with the average case of string matching algorithms. The string matching problem consist in find all occurrences (or the first occurrence) of a pattern in a text, where the pattern and the text are strings over some alphabet. 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 [16]. However, between two different algorithms the constant in the linear term can be very different. For example, the constant multiple in the Naive algorithm is  $m$ , where as for the Knuth-Morris-Pratt algorithm it is 2. On the other hand, the average case of this algorithms is not well studied. All previous analyses assumes an approximate model or are not formally developed.

---

\*This work was also supported by the Institute for Computer Research of the University of Waterloo and the University of Chile, Santiago, Chile.

We derive an exact analysis for the Naive algorithm, the Rabin-Karp algorithm (a probabilistic one) and for different variants of the Boyer-Moore algorithm in a random text model. An approximate analysis is derived for the Knuth-Morris-Pratt algorithm. Extensive experimental results support these analysis. Also, these results are compared against experimental results in real text.

For patterns of length 2, we provide an exact analysis of almost all known algorithms. We also include some optimal algorithms in different senses for this case.

Based on the experimental results for real text, we present a variation of the Boyer-Moore algorithm that improves the search time for non-uniform text, based on statistics previously obtained from the text.

## 2 Previous Results

Not too many papers mention the average case of string matching algorithms. Moreover, only one paper deals exclusively with this case. The following is a review of previous work on this topic.

The first paper is the classic Knuth, Morris and Pratt algorithm, published in 1977 [14]. This work presents the first algorithm, in 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 a preprocessing of the pattern that takes  $O(m)$ . In fact, the expected number of comparisons performed by this algorithm (search time only) is bounded by

$$n \leq \bar{C}_n \leq 2n .$$

In the same paper, it is showed that there exists an algorithm in which the expected worst case number of inspections <sup>1</sup> is  $O(\frac{\log_c m}{m}n)$  in random text, where  $c$  is the size of the alphabet. Is conjectured that this is also a lower bound. For some patterns, the expected number of characters examined by this algorithm is  $O(\frac{n}{m})$ . Also, an optimal algorithm for patterns of length 2, is presented. The algorithm is optimal in the sense that it minimizes the average number of characters examined to find all the occurrences of the pattern in the text. The expected number of inspections is

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

That same year, another classic algorithm was published by Boyer and Moore [4]. The main idea, is to search from right to left in the pattern. With this schema, the search is faster in average. An analysis of the algorithm in

---

<sup>1</sup>One character can be compared many times, but inspected only once.



the average 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 [19] published the first paper on the complexity of the string matching problem. He proved the conjecture raised in Knuth, Morris and Pratt's paper, showing that the minimum average number of characters that need to be examined in a random string of length  $n$  is

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

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.

Horspool in 1980 [13] presented a simplification of the Boyer-Moore algorithm, and based on empirical results shows that this simpler version is as good as the original Boyer-Moore algorithm. Moreover, the same results shows 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 use a hardware instruction to scan for the character with lowest frequency present in the pattern), for patterns of length greater than 5.

The first (published) analysis of the Naive algorithm appears in 1985 by Barth [3]. This work uses Markov chains, assuming that the probabilities of 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

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

where  $c$  (as usual) is the alphabet size. This paper, using an heuristic model, also derives an approximation for the Knuth-Morris-Pratt algorithm, given by

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

With this results, Barth states the following approximation for the ratio of the expected number of comparisons between the Knuth-Morris-Pratt and the Naive algorithms:

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

### 3 Preliminaries

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. It might be argued that the average case taken over random strings is of little interest, since a user rarely search 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.

#### 3.1 Random Text

Let  $\Sigma$  be the alphabet of size  $c$ . Let  $f_i$  be the probability of choosing the character  $i$  from the alphabet. If you choose two characters independently from the alphabet, the probability that these characters are equal is

$$p_{\text{equal}} = \sum_{i=1}^c f_i^2$$

We define a *random string* of length  $l$  as a string built by concatenation of  $l$  characters chosen independently from  $\Sigma$  uniformly (that is  $f_i = \frac{1}{c}$  for all  $i$ ). In the following we will use random strings, however almost all the results can be translated to the general case by changing  $1/c$  to  $p_{\text{equal}}$ .

Let  $\text{text} = t_1 t_2 \dots t_n$  be a random string of length  $n$  and let  $\text{pattern} = p_1 p_2 \dots p_m$  be a random string of length  $m$  with  $n \geq m$ . In practice, we will use  $n \gg m$ .

**Lemma 3.1** *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:** Follows by definition, because each letter is chosen independently, and so these events are uncorrelated. ■

Is possible to extend the previous lemma, to the case of a memoryless algorithm. In that case, the result also holds, because 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 3.2** *The probability of finding a match between a random text of length  $n$  and a random pattern of length  $m$  is*

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

**Proof:** We have

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

using the previous lemma. ■

**Lemma 3.3** *The expected number of comparisons to decide if a random pattern of length  $m$  match or not with a random text of length  $m$  is*

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

**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. Then, using the first lemma, we have that

$$\bar{C}_m = 1 + \sum_{i=1}^{m-1} 1 \times \text{Prob}\{t_1 = p_1 \wedge \cdots \wedge t_i = p_i\} = \sum_{i=0}^{m-1} \frac{1}{c^i} = \frac{c}{c-1} \left(1 - \frac{1}{c^m}\right)$$

■

**THEOREM 3.1** *The expected number of matches of a random pattern of length  $m$  in a random text of length  $n$  is*

$$E[\text{matches}] = \frac{n - m + 1}{c^m} \quad \text{if } n \geq m,$$

*otherwise ( $n < m$ ) is 0.*

**Proof:** In each  $m$  consecutive positions in the text, the probability of finding a match is given by  $1/c^m$ . Each one of this substrings is independent from the others by Lemma 3.1, and there are  $n - m + 1$  possible substrings that could match with the pattern. ■

In many cases, we will 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 [5]

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

where  $x_{\min}$  and  $x_{\max}$  denotes the range of the random variable  $x$ . If, the variance of  $x$  converges to 0, asymptotically in some parameter, then the lower bound is an equality (by the law of large numbers).

### 3.2 Markov Chains

We will need some basic results from Markov chains. A stochastic process is a Markov process if the probability of one event only depends 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  $t$  to time  $t + 1$  depends on the state at time  $t$  and the final state at time  $t + 1$  [6].

Let  $S = \{s_1, \dots, s_r\}$  be the possible states in the chain, such that  $r$  is finite. The *transition matrix* of the process is defined as

$$\mathbf{P} = [p_{ij} = \text{Prob}\{i \rightarrow j\}]_{r \times r},$$

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

$$\mathbf{p}^{(t)} = \mathbf{p}^{(0)} \mathbf{P}^t \quad (n = 1, 2, \dots)$$

where  $\mathbf{p}^{(0)}$  is the vector of initial 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 other state. Then, if  $\pi$  is the stationary vector of state occupation probabilities, that is

$$\lim_{t \rightarrow \infty} \mathbf{p}^{(t)} = \pi$$

then,  $\pi$  is the solution the following linear system of equations [6]

$$(\mathbf{P} - \mathbf{I})\pi = 0 \quad , \quad \sum_j \pi_j = 1$$

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

### 3.3 Experimental results

The empirical data, in almost all the algorithms, consists of results for two types of text: random text and real text. In both cases, 100 runs were performed, obtaining all the statistical measures with their 95% confidence interval. In each case, patterns from length 2 to length 15 were considered. The two main costs measured, were the number of comparisons performed between a character in the text and a character in the pattern, and the number of instructions executed. The instruction model was as simple as possible, considering any instruction to have cost 1 unit, except multiplication,

division and modulus which cost 2 units. In the graphs  $Ibarsubn$  denotes the number of instructions performed while searching a text of length  $n$ .

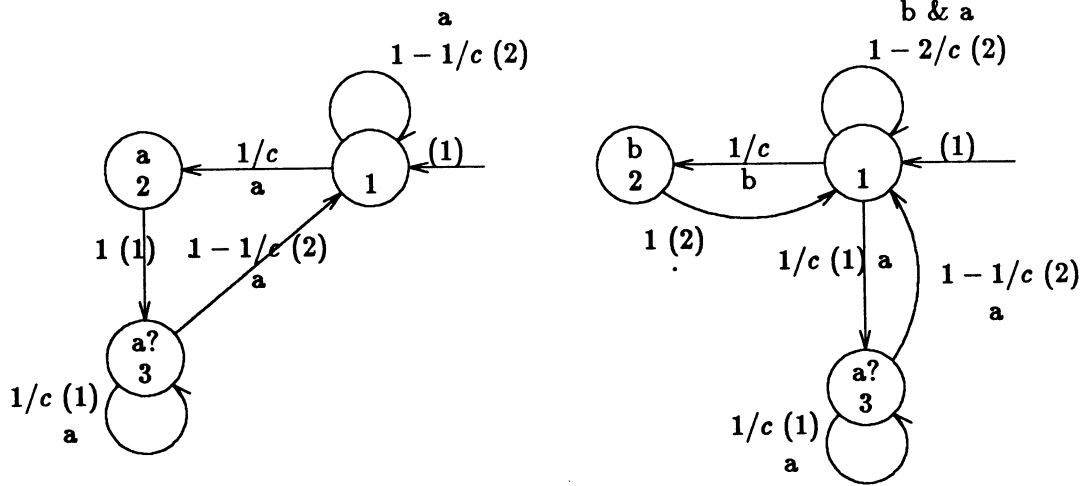
In the case of random text, the text was of length 20000, and both the text and the pattern were chosen at random uniformly from an alphabet of size  $c$ . The values 2 (binary), 4 (DNA), 10, 30 (lower case English) and 90 were considered. The results are more sensitive to the random number generator for small alphabets. Moreover, when an algorithm work differently for each pattern (for example Knuth-Morris-Pratt for alphabet size 2).

In the case of real text, we used a document in English. The document selected was the Constitution of The United States. The length of this text is 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 (or sentence). The alphabet used, was the set of lower case characters, some numbers and some punctuation symbols, giving 33 characters.

## 4 Optimal Algorithms

All the linear algorithms that will be presented later, are optimal time algorithms in the worst case. However, they are not space optimal in the worst case, because these linear algorithms use space that depends on the size of the pattern or the size of the alphabet, or both. Galil and Seiferas [8,10] show that is possible to have linear time worst case algorithms using constant space. Also they show that the delay between two characters in the text is constant. That is, it is possible to search in real time [9].

Yao's result [19] is a lower bound for the average case number of inspections. What happens with the average case for the number of comparisons? First, we will find an optimal algorithm on the number of inspections for a pattern of length 2 [14] using Markov chains. The following diagram shows the Markov chains for the two possible patterns (two equal characters or two different characters). Note, that the pattern is scanned right to left.



The algorithm is optimal in the sense that it uses all the information from the past, and then, it inspect the minimum possible number of characters in the average. Note that we have three labels in each edge. One, is the value of the last character inspected ("," means any character), the other is the probability that this transition happens, and the last (between parenthesis) is how many positions we can advance in the text. Using, the results from Markov chains we have that for the pattern "aa"

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

and that

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

Now, we can compute, the expected number of characters shifted per inspection (for each new transition we inspect a new character), or  $csi$  for short.

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

For the case "ab" the result is the same!. Then, using the Kantorovich inequality (in this case an equality) we have Knuth's result:

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

The expected number of comparisons of this algorithm is not the same. The reason is that for the case "ab", in the state 1, we need two comparisons to decide which is the new state and in state 3, we need other comparison to

decide if we report a match or not (unless  $c = 2$ ). Arranging the comparisons to minimize the number of times that the second comparison is performed and using a different code for each case, we have an algorithm that performs less comparisons on the average than Knuth's algorithm. Table 1 shows the empirical results for both algorithms (patterns of length 2). Both algorithms are optimal on the expected number of comparisons for  $c = 2$ .

Algorithm	$c = 2$	$c = 4$	$c = 10$	$c = 30$	$c = 90$
Knuth	1.1716	1.1086	1.04762	1.01654	1.00551
Ours	0.969	0.972	0.983	0.997	1.0005

Table 1:  $\frac{\bar{C}_n}{n-1}$  for optimal algorithms

**Lemma 4.1** *The expected number of comparisons performed by the previous algorithm to search a pattern of length 2 in a text of length  $n$  is*

$$\frac{2c(c^2 + c - 1)}{2c^3 + 3c^2 - 4c + 1} \leq \bar{C}_n \leq \frac{9c(c^2 + c - 1)}{4(2c^3 + 3c^2 - 4c + 1)}$$

**Proof:** Computing the extra comparisons from the probability vector and using the Kantorovich inequality ( $C_n$  ranges from 1 to 2). ■

Is the previous algorithm optimal on 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 8) for the pattern “ab” performs less comparisons on the average for all  $c > 2$ . Namely

$$\frac{c(c+1)(c^2 + c - 1)}{(2c - 1)(c^3 + c^2 - c + 1)} \leq \bar{C}_n \leq \frac{9c(c+1)(c^2 + c - 1)}{8(2c - 1)(c^3 + c^2 - c + 1)}$$

An interesting question is which is the number of states of the deterministic finite automata that represents a general optimal algorithm that scan the pattern right to left (or a “Boyer-Moore dfa”). An obvious upper bound is  $2^m$ . In [14], it is explained why a pattern consisting in  $m$  different symbols requires  $O(m^2)$  states. An  $O(m^3)$  lower bound over a three character alphabet is known [11]. Is easy to see from the Knuth-Morris-Pratt algorithm, that an optimal algorithm that scan the pattern *left to right* needs also at least  $O(m^2)$  (and possibly not more).

Using the same approach, it is possible to obtain  $\bar{C}_n$  for any algorithm, for a pattern of fixed small length. Table 2 shows the results for patterns of length 2 in function of the alphabet size  $c$  for Boyer-Moore type algorithms (see section 8).

Algorithm	Lower Bound	Upper Bound
Boyer-Moore [4]	$\frac{c+1}{2c-1}$	$\frac{25(c+1)}{16(2c-1)}$
Boyer-Moore-Galil [7]	$\frac{c^3}{2c^3-3c^2+3c-1}$	$\frac{9c^3}{8(2c^3-3c^2+3c-1)}$
Simplified Boyer-Moore	$\frac{c^2(c+1)}{2c^3-c^2-2c+2}$	$\frac{25c(c+1)}{16(2c^2-2c+1)}$

Table 2: Lower bound on  $\frac{\bar{C}_n}{n-1}$  for Boyer-Moore type algorithms

## 5 Naive Algorithm

The naive or brute force algorithm, is the simplest string matching method. The idea consist in try to match any substring of length  $m$  in the text with the pattern. Clear, this is a memoryless algorithm, because the only information recorded, is how many characters are matching at each point, but we forgot all of that in the next trial.

**THEOREM 5.1** *The expected number of comparisons text-pattern performed by the naive or brute-force algorithm to search a pattern of length  $m$  in a text of length  $n$  ( $n \geq m$ ) is*

$$\bar{C}_n = \frac{c}{c-1} \left(1 - \frac{1}{c^m}\right) (n - m + 1)$$

**Proof:** Using Lemma 3.3 and Theorem 3.1 we have that there are  $n - m + 1$  substrings, and in each one of them, we perform on the average  $\bar{C}_m$  comparisons. ■

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

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

This is drastically different from the worst case ( $mn$ ). Figure 1 shows the theoretical curve for some values of  $c$ , along with the empirical results. The agreement between both curves is almost exact.

## 6 Knuth-Morris-Pratt Algorithm

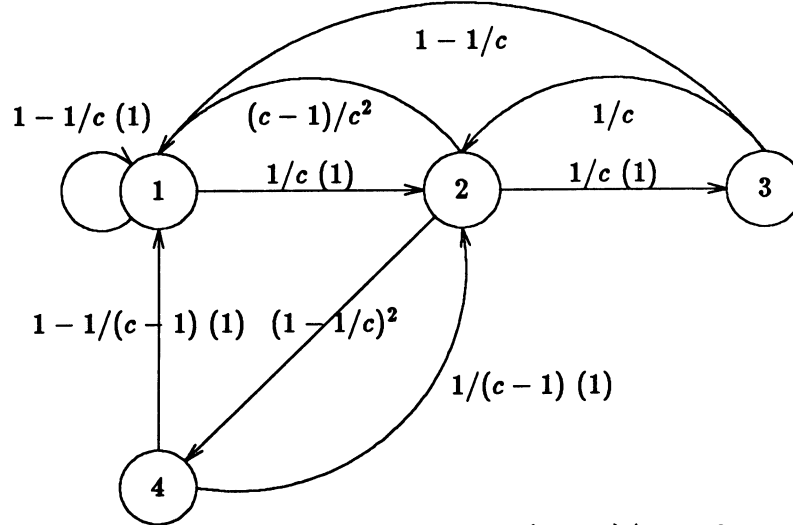
The basic idea behind this algorithm is that each time a mismatch is detected, our “false start” consist in characters that we know in advance. Then, we can take advantage of this information instead of backing up over all those known characters. Moreover, is always possible to arrange things



so that the pointer in the text is never decremented. For this, the algorithm preprocesses the pattern to obtain a table that gives to which state of the match you need to go if we found a mismatch. Further explanations can be found in the original paper [14] or in text books (for example [18]). In the worst case, the time is  $2n + O(m)$ .

The basic idea of this algorithm, also was discovered independently by Aho and Corasick [1], to search for a set of patterns. However the space used and the preprocessing time to search for one string is improved in Knuth-Morris-Pratt algorithm. A variation that computes the next table as needed when we are searching is presented in [2].

The Knuth-Morris-Pratt algorithm is not a complete memoryless algorithm. In fact the next table acts as different states during the search. The fact that Knuth-Morris-Pratt algorithm is not memoryless, makes Barth model invalid and then his results [3]. If we have that two characters are different, the next state is such that we have a new partial match and that the new character in the pattern is *different* from the old one. Then, the Markov chain for the case  $m = 2$  has one state more for each different next:



In general the number of states will be  $m(m+1)/2 + 1$  for a pattern of length  $m$  and we compute the following exact results:

**Lemma 6.1** *The expected number of comparisons text-pattern performed by the Knuth-Morris-Pratt algorithm to search in a text of length  $n$  ( $n \geq m$ ) is*

$$1 + \frac{1}{c} - \frac{1}{c^2} + \frac{1}{c^3} + O(1/n) \leq \frac{\bar{C}_n}{n} \leq \frac{9}{8} \left( 1 + \frac{1}{c} - \frac{1}{c^2} + \frac{1}{c^3} \right) + O(1/n)$$

for a pattern of length 2 and

$$\frac{c^5 + c^4 - c^3 + 1}{c^5 - c + 1} + O(1/n) \leq \frac{\bar{C}_n}{n} \leq \frac{9(c^5 + c^4 - c^3 + 1)}{8(c^5 - c + 1)} + O(1/n)$$

for a pattern of length 3.

**Proof:** Computing the steady state probability as in section 4 and using the Kantorovich inequality ( $C_n$  ranges from 1 to 2 for any pattern length).

■

If we suppose that the memoryless model is valid, the result is not far away from the exact result, and is asymptotically similar in the size of the alphabet. Then, we can approximate the Knuth-Morris-Pratt algorithm using this model.

In the general case, the Markov chain will have  $m + 1$  states. In state  $j$  we have the following transitions:

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

Then, the expected number of characters skipped per comparison is

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

Now, we can made the following approximation:

$$Prob\{next[j] = i/mismatch\} \approx \frac{(c - 1)^{j-i}}{c^{j-1}}$$

for  $1 \leq i < j$ , and  $Prob\{next[j] = 0/mismatch\}$  is the complementary probability. This approximation is computing the probability that the last  $i - 1$  characters matched are a prefix of the pattern (that is match the beginning of the pattern) and that the other characters must be different from the character in the pattern that mismatch. Computing an asymptotic approximation for  $Prob\{next[j] = 0\}$  and using that  $\sum_{j=2}^m p_j = 1 - p_1 - p_{m+1} = 1 - p_1 + O(c^{-m})$  we obtain an asymptotic approximation for the expected number of comparisons

$$\bar{C}_n \approx 1 + \frac{1}{c} - \frac{1}{c^2} + O(c^{-3})$$

and is tight for alphabets of size greater than 10, as can be seen from the empirical results. Figure 2 show the empirical results contrasted with the approximation developed here. Barth approximation of Knuth-Morris-Pratt algorithm is not good (too optimistic), as shown in Figure 2 for the case  $c = 4$ . In fact, the average ratio between the two algorithms is better approximated by

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

and then both algorithms are closer on the average for large alphabets.

Table 3 shows the empirical results contrasted with the exact theoretical values for  $m = 3$ .

Result	$c = 2$	$c = 4$	$c = 10$	$c = 30$	$c = 90$
Theoretical	1.322	1.1919	1.090	1.032	1.01098
Empirical	1.321	1.186	1.0896	1.03343	1.01119

Table 3: Results on  $\frac{C_n}{n-1}$  for the Knuth-Morris-Pratt algorithm

## 7 Rabin-Karp Algorithm

Another approach to string searching is to use hashing. All that we need to do is to compute the hash function of each possible  $m$ -character substrings of the text and check if it is equal to the hash function of the pattern. The problem with this method is that it seems at first to be just as hard as in the Naive algorithm, and using more memory for the table.

However, Rabin and Karp [15] found an easy way to get around this problem for the hash function  $h(k) = k \bmod q$  where  $q$  (the table size) is a large prime. Their method is based on computing the hash function for position  $i$  in the text given it value for position  $i - 1$ . With this the algorithm takes time proportional to  $n + m$  in almost all the cases, without using more memory. Note that it really finds a position in the text which has the same value as the pattern, so, to be sure, we really should do a direct comparison of that text with the pattern. However, the use of a large value of  $q$  makes a collision extremely unlikely. Theoretically, this algorithm could still take  $mn$  steps on the worst case, if we check and we have too many matches or collisions. In all our empirical results we observed 3 collisions in  $10^7$  computations of the hash function (generally, for big alphabets). Then, in practice, we can forget the checking, obtaining a truly linear algorithm.

The hash function used, is writing a string as a base- $d$  number, where  $d$  is the number of possible characters. More easy, is to use  $d$  a power of 2,

such that  $d \geq c$ . In this case, all 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. For more details of the algorithm see [18].

**THEOREM 7.1** *The expected number of comparisons pattern-text performed by the Rabin-Karp algorithm to search a pattern of length  $m$  in a text of length  $n$  is asymptotically on  $n$  and  $m$  (with  $n \gg m$ )*

$$\frac{\bar{C}_n}{n} = \lambda + \frac{m}{c^m} + O\left(\frac{m}{q}\right)$$

where  $\lambda$  is the cost of computing the hash function expressed as comparison units.

**Proof:** We have that

$$\bar{C}_n = \lambda(n + m) + \frac{m(n - m + 1)}{c^m} + mn \text{Prob}\{\text{collision}\}$$

where the first term is the computation of all the hashing functions and the second term is the number of comparisons used to check all the matches found on average. The last term, depends on the number of collisions. The probability of a collision is proportional to  $1/q$  with the assumption that the hashing function is uniform. ■

The empirical results are plotted using  $\lambda = 1$  in figure 3. They agree very well for almost all the alphabet sizes.

## 8 Boyer-Moore Algorithm

A practical improvement on the average time is obtained searching from the *right to the left* in the pattern [4]. The algorithm repeatedly positioning the pattern over the text and attempting to match it. For each positioning that arises, the algorithm starts matching the pattern against the text from the right end of the pattern. If no mismatch occurs, then the pattern has been found. Otherwise the algorithm computes a shift; that is, an amount by which the pattern will be moved to the right before a new matching attempt is undertaken.

The shift is computed using two heuristics. The *match* heuristic is based on the idea that when the pattern is moved to the right, it has to (1) match over all the characters previously matched, and (2) bring a *different* character over the character of the text that caused the mismatch. Secondly, the *occurrence* heuristic uses the fact that we must bring over the character of the text that caused the mismatch, the first character of the pattern that will match it.

Both shifts can be obtained from precomputed tables based solely on the pattern and the alphabet used. Hence, the space needed is  $m + c + O(1)$ . Given these two shifts, the algorithm chooses the largest one. The shift strategy also can be applied after a match.

Knuth [14] show 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 too many matches (for example, the case of an small alphabet). An alternative proof, more simple, can be found in [12]. In [14] the preprocessing of the pattern is showed to be linear in the size of the pattern, as the Knuth-Morris-Pratt algorithm. However, that algorithm is incorrect. The corrected version can be found on Rytter's paper [17].

The simulations results obtained here agrees well with the empirical and theoretical results on the original paper [4]. See figure 4 for the empirical and theoretical results of our paper. Here, we go further in the analysis, to obtain asymptotic results.

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

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

For the match heuristic, given a mismatch at position  $k+1$  from the right, the shift is of size  $k$ , if the substring matched, 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 at 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 [4]

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

with

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

The solution for the recursion is

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

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

$$P_j(k) = O_{c_j}(k) \sum_{i=1}^j M_i(k) + M_j(k) \sum_{i=1}^{j-1} O_{c_i}(k) .$$

Hence, the average number of characters skipped per comparison is

$$E[csc] = \sum_{k=0}^{m-1} (1 - 1/c) \frac{(1/c)^k}{k+1} \sum_{j=1}^m j P_j(k) + \frac{1}{mc^m}$$

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

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

To improve the worst case, Galil [7] modifies the algorithm, remembering how many overlapping characters we can have between two successive matches. Then, instead of going from  $m$  to 1, the algorithm goes from  $m$  to  $l$  where  $l$  ( $\geq 1$ ) depends if the last event was a match or not.

This algorithm is truly linear, with a worst case  $O(n + m)$ . However, according to the empirical results (figure 3), as expected, only improves the average case for small alphabets, at the cost of using more instructions. The improvement is of  $O(c^{-m})$ .

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

Then, each time that a mismatch is found, we select the maximum from the heuristic table or 1, for the absolute shift of the pattern. This algorithm, because we are not using the *match* heuristic, is memoryless, and then we can apply the first lemma of section 3.

**Lemma 8.1** *The expected value of each shift of the pattern over the text on the Simplified Boyer-Moore algorithm is*

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

*and the expected number of character skipped per comparison is*

$$E[csc] = (c-1)(1 - c(1 - 1/c)^m) \ln\left(\frac{c}{c-1}\right) + c(c-1) \ln\left(\frac{c^2}{c^2 - c + 1}\right) + O\left(\frac{1}{mc^m}\right)$$

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

$$\bar{S} = \frac{1}{c^m} + \sum_{k=0}^{m-1} (1 - 1/c)(1/c)^k \sum_{j=1}^m j O_{c_j}(k)$$

giving the desired result. For the expected number of character skipped per comparisons we take the same average weighted by the number of comparisons of each case, that is

$$E[csc] = \frac{1}{mc^m} + \sum_{k=0}^{m-1} (1 - 1/c) \frac{(1/c)^k}{k+1} \sum_{j=1}^m j O_{c_j}(k)$$

We use the fact that  $\sum_{k=0}^{\infty} x^{k+1}/(k+1) = -\ln(1-x)$  to approximate the expression above. ■

**THEOREM 8.1** *The expected number of comparisons text-pattern performed by the Simplified Boyer-Moore algorithm to search a pattern of length  $m$  in a text of length  $n$  ( $n \geq m$ ) is asymptotically on  $n$  and  $m$  (with  $m \ll n$ )*

$$\frac{\bar{C}_n}{n} \geq \frac{1}{(c-1) \left( \ln\left(\frac{c}{c-1}\right) + c \ln\left(\frac{c^2}{c^2 - c + 1}\right) \right)} + O((1 - 1/c)^m)$$

*and asymptotically on  $n$  and  $c$  (with  $c \ll n$ ) is*

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

**Proof:** The expected number of comparisons is bounded by  $\frac{n-m+1}{E[csc]}$ . ■

Figure 5 shows the theoretical and empirical results for this case. An interesting fact to improve the previous algorithm, is to note, that after we know that the pattern matches (or not), any of the characters from the text can be used to address the heuristic table. Based in that, Horspool [13]

improves the algorithm, using always the character in the text corresponding to the last character of the pattern. To avoid the comparison to know the maximum in the case that the value of the table is 0 (last character of the pattern), he notes that is possible to define initially the entry in the occurrence table for the last character in the pattern as  $m$  and then compute the heuristic table only up to the first  $m - 1$  characters of the pattern. The code in the C programming language for an optimized Horspool version of the Boyer-Moore algorithm is really simple. Namely

```

for( k=0; k<MAXSYM; k++ ) d[k] = m;      /* preprocessing          */
for( k=1; k<m; k++ ) d[pat[k]] = m-k;
pat[0] = CHARACTER_NOT_IN_THE_TEXT;      /* to avoid having          */
text[0] = CHARACTER_NOT_IN_THE_PATTERN; /* special code if the
*/
/* pattern matches the
*/
k = m;                                  /* beginning of the text
*/
while( k <= n )
{
    k1 = k;
    for( j=m; text[k1] == pat[j]; j-- ) k1--;
    if( j == 0 ) Report_match_at_position( k1 + 1 );
    k += d[text[k]];
}
/* restore pat[0] and text[0] if necessary */

```

The simplicity of the Horspool version, also simplifies the analysis.

**Lemma 8.2** *The expected value of each shift of the pattern over the text (that is the expected value of  $d[\text{text}[k]]$ ) is*

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

**Proof:** We have that 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  $1/c(1 - 1/c)^{m-j-1}$ . Then

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

■



**THEOREM 8.2** *The expected number of comparisons text-pattern performed by the Horspool version of the Simplified Boyer-Moore algorithm to search a pattern of length  $m$  in a text of length  $n$  ( $n \geq m$ ) is*

$$\bar{C}_n \geq \frac{1 - \frac{1}{c^m}}{(c-1) \left(1 - \left(1 - \frac{1}{c}\right)^m\right)} (n - m + 1)$$

*and asymptotically on  $n$  and  $m$  (with  $m \ll n$ ) is*

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

*and asymptotically on  $n$  and  $c$  (with  $c \ll n$ ) is*

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

**Proof:** The expected number of times that we shift the pattern is bounded by  $\lfloor \frac{n-m+1}{s} \rfloor$ . Each time, on the average, we perform  $\bar{C}_m$  comparisons. This expression is valid in this case, because the expected shift does not depend in which position the pattern did not match. It is possible to obtain a lower bound using  $E[csc]$ , but the result is weaker than the previous one. ■

Based on the analysis, this version of the simplified Boyer-Moore algorithm is better, and is as good as the original Boyer-Moore algorithm for alphabets of size 10 or bigger. Figure 6 shows the empirical results and the theoretical results for Horspool version.

Finally, figure 7 shows the expected number of instructions per character for all the algorithms ( $c = 30$ ). Based on the empirical results, it is clear, that Horspool variant is the best known algorithm for almost all pattern lengths and alphabet sizes.

## 9 Searching on Real Text

Figure 8 shows the same empirical results of previous sections for a real text (see section 3.3). Although the results are very similar, the main difference between random text and real text is the following: if we have a partial match, the probability that the next character will match, will be generally greater than  $1/c$ . For example, suppose that we are searching for *computer* and at some point we have *compu* as a partial match. Which is the probability that the next character is a *t*?. The answer depends on how many words in English begin with that prefix, but obviously is greater than  $1/30$ . This tell us that in real text the value of  $\bar{C}_m$  is a little higher.

Which is the model for this correlation when we have partial matches?. Suppose that the probability of match the first character of the pattern is

$p_{equal}$  (like random text). And then, each time, if we have a partial match the probability increases up to some point.

Based on the frequency of the characters in the real text used, we found that  $p_{equal} = 0.073$  and if we do not include spaces as a valid character for the first letter of a word, the value is  $p_{equal} = 0.046$ .

Figures 9 and 10 shows what happens in practice for four algorithms: The naive and Knuth-Morris-Pratt algorithms (left to right) and the Boyer-Moore and Boyer-Moore-Horspool algorithm (right to left). The results are plotted for 4 different pattern lengths.

Note that for the left to right algorithms  $p_{equal}$  is near 0.046 as expected. This is because only prefix of words (sentences) are used. Other interesting fact, is that the probabilities are more or less the same for both algorithms (the naive algorithm try all positions, Knuth-Morris-Pratt not).

For the right to left algorithms position 1 means a mismatch in position  $m$  and so on (reverse order). Here, we have that  $p_{equal}$  is near 0.073, because the prefix of a sentence can finish with a space (likely on long patterns). Again, the probabilities for both cases are similar.

Based, on the results, a possible model for this probability, is an exponential curve

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

where  $\alpha$  is a parameter that control how fast the curve converges to 1. We found that a good approximation for the left to right algorithms is

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

and for the right to left algorithms is

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

Both curves are also plotted in figures 9 and 10. Is interesting to see that the correlation in both directions and for all values of  $m$  is very similar. These experiments are not conclusive, and is necessary to run more general simulations. However, it is clear that the correlation is very high, and then to search on real text is *slightly hard* than to search on random text.

How we can improve the average time on real text?. Based only on the pattern, it is not possible to improve the length of the expected shift on 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”. May be good idea to compare first if ‘z’ is in the text rather than to compare ‘e’, because ‘e’ is a character with high frequency in English. Then, the optimal comparison order is giving by the probability of finding each character on the text. We can find that information preprocessing the text, or using previous statistics about text in the language used.

Also, is possible at the same time, to compute dynamically the maximum value of the shift for all the characters compared in one trial. However, this idea is not good, because we increase the number of instructions per comparison more than the final savings. Other way, if we cannot determine which is the best character in the text to compute the shift, is to use an heuristic based on the pattern. One possible heuristic is to use the position  $k - (m - j^{opt})$  on the text to compute the shifts, with  $j^{opt}$  given by

$$j^{opt} = \max(j(1 - Prob\{p_j\}), j = 1..m)$$

This is based on the fact that if the character in the text mismatch with  $p_j$ , the absolute shift is  $j$  if the character is not in the pattern. The position  $j^{opt}$  will be different from  $m$  if we have two of the following cases:

- a long pattern
- non uniform text
- a character with high frequency

The main drawbacks of the algorithm is that we need  $O(m \log m)$  pre-processing time and  $O(m + c)$  space. Also, in the internal loop we have 3 instructions more per comparison, and the heuristic fails many times, because the character in the text can mismatch, but is in the pattern. A simplified version of this heuristic that optimizes both things only in the last positions without increasing the number of instructions improves Horspool version for long patterns in real text. This improvement is greater if the probability distribution of the symbols is further away from the uniform distribution.

## 10 Conclusions

Using a simple random model for text, we have analyzed almost all the known algorithms for string matching. The empirical results show that this model is close enough to real text (being searching in real text slightly harder) to predict the average behaviour of these algorithms. More over, the empirical results and the theoretical results are almost similar for typical alphabets ( $c \geq 10$ ).

Asymptotically on  $m$ , the length of the pattern, the constant factor in the linear term of the average number of comparisons is given by a function that only depends on the alphabet size (that is, do not converge to 0 with large  $m$  [19]) for almost all the known algorithms. Analogously, asymptotically on  $c$ , the size of the alphabet, the constant factor depends only on the pattern size.

Up to now, Horspool version of the Boyer-Moore algorithm is the best algorithm, according to the average number of machine instructions performed, for almost all pattern lengths. For non-uniform text the variant presented here may be better, depending on the probability distribution of the symbols used.

## References

- [1] Aho, A. and Corasick, M. "Efficient String Matching: An Aid to Bibliographic Search", *Communications of the ACM*, 18 (1975), 333-340.
- [2] Barth, G., "An Alternative for the Implementation of Knuth-Morris-Pratt Algorithm", *Information Processing Letters* 13 (1981), 134-137.
- [3] Barth, G. "Relating The Average-Case Costs of the Brute-Force and Knuth-Morris-Pratt String Matching Algorithms", in NATO ASI Series, Vol F12, 45-58, *Combinatorial Algorithms on Words*, Edited by A. Apostolico and Z. Galil, Springer-Verlag, 1985.
- [4] Boyer, R. and Moore, S. "A Fast String Searching Algorithm", *Communications of the ACM*, 20 (1977), 762-772.
- [5] Clausing, A. "Kantorovich-Type Inequalities", *The American Mathematical Monthly* 89 (1982), 314-330.
- [6] Cox, D. and Miller, H. "The Theory of Stochastic Processes", Chapman and Hall, London, 1965.
- [7] Galil, Z. "On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm", *Communications of the ACM*, 22 (1979), 505-508.
- [8] Galil, Z. and Seiferas, J., "Saving Space in Fast String-Matching", *SIAM Journal of Computing* 9 (1980), 417-438.
- [9] Galil, Z. "String Matching in Real Time", *Journal of the ACM* 28 (1981), 134-149.
- [10] Galil, Z. and Seiferas, J. "Time-space-optimal string matching", *Journal of Computer and System Sciences* 26 (1983), 280-294.
- [11] Galil, Z. "Open Problems in Stringology", in NATO ASI Series, Vol F12, 1-8, *Combinatorial Algorithms on Words*, Edited by A. Apostolico and Z. Galil, Springer-Verlag, 1985.

- [12] Guibas, L. and Odlyzko, A. "A New Proof of the Linearity of the Boyer-Moore String Searching Algorithm", *SIAM Journal of Computing* 9 (1980), 672-682.
- [13] Horspool, N. "Practical Fast Searching in Strings", *Software-Practice and Experience* 10 (1980), 501-506.
- [14] Knuth, D., Morris, J. and Pratt, V. "Fast Pattern Matching in Strings", *SIAM Journal of Computing*, 6 (1977), 323-350.
- [15] Rabin, M. and Karp, R. "Efficient Randomized Pattern-Matching Algorithms", manuscript.
- [16] Rivest, R. "On the Worst-Case Behavior of String-Searching Algorithms", *SIAM Journal of Computing*, 6 (1977), 669-674.
- [17] Rytter, W. "A Correct Preprocessing Algorithm for Boyer-Moore String-Searching", *SIAM Journal of Computing*, 9 (1980), 509-512.
- [18] Sedgewick, R. *Algorithms*, Addison-Wesley, Reading, Mass., 1983.
- [19] Yao, A. C. "The Complexity of Pattern Matching for A Random String", *SIAM Journal of Computing*, 8 (1979), 368-387.

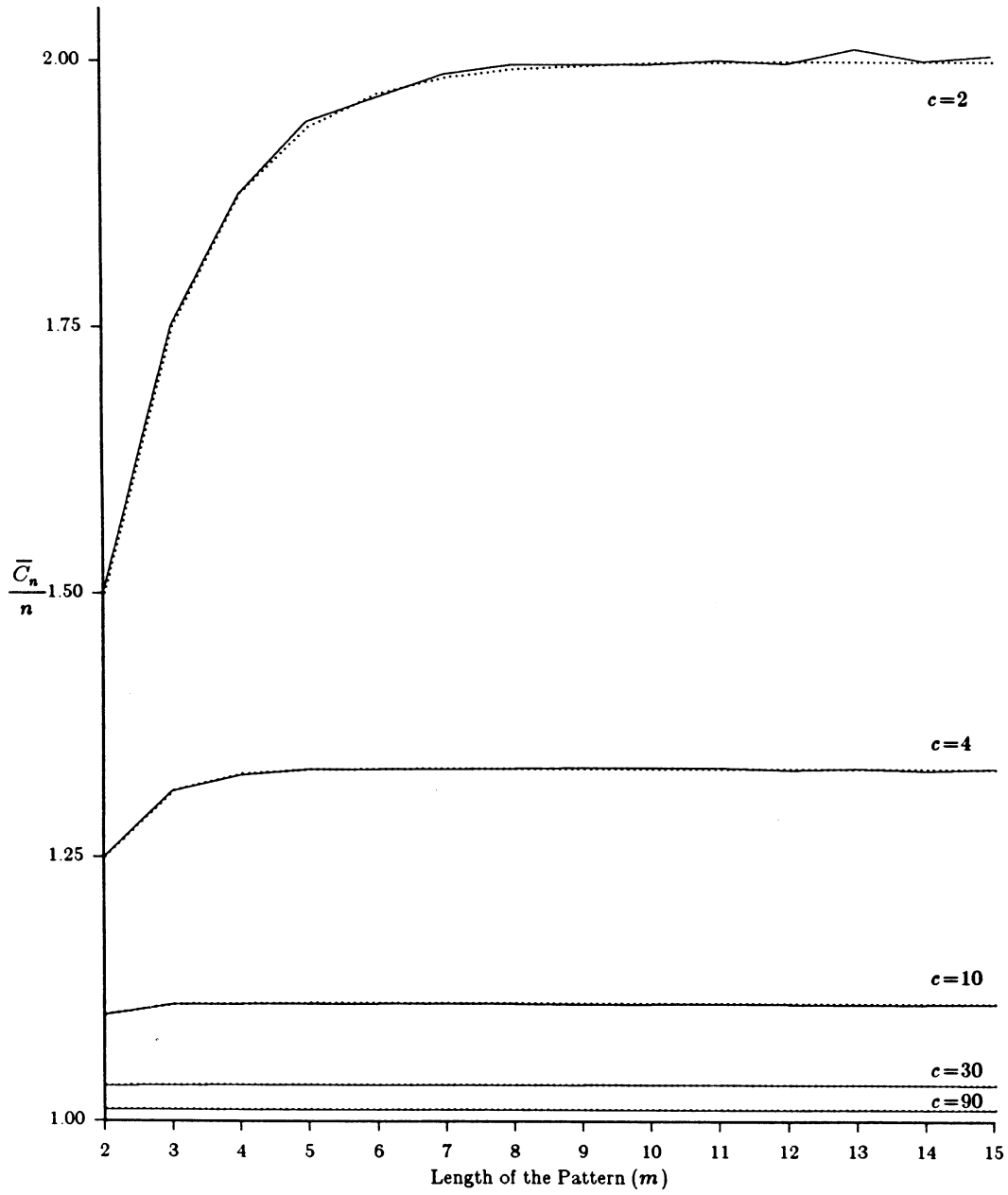


Figure 1. Simulations results for the naive algorithm in random text  
(dotted line show the theoretical results)

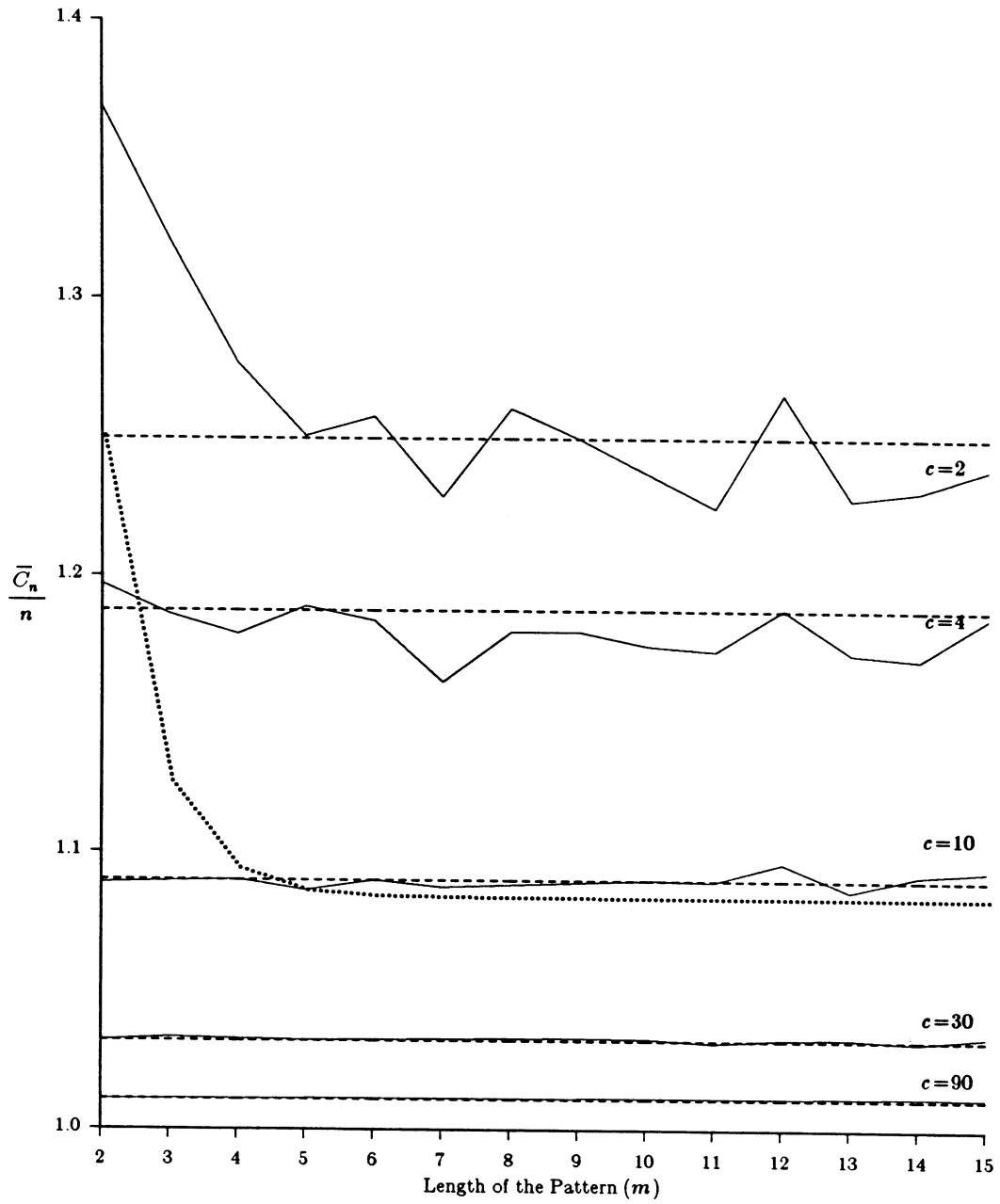


Figure 2. Simulations results for the Knuth-Morris-Pratt algorithm in random text (dashed line = theoretical approx, dotted line = Barth approx. for  $c=4$ )

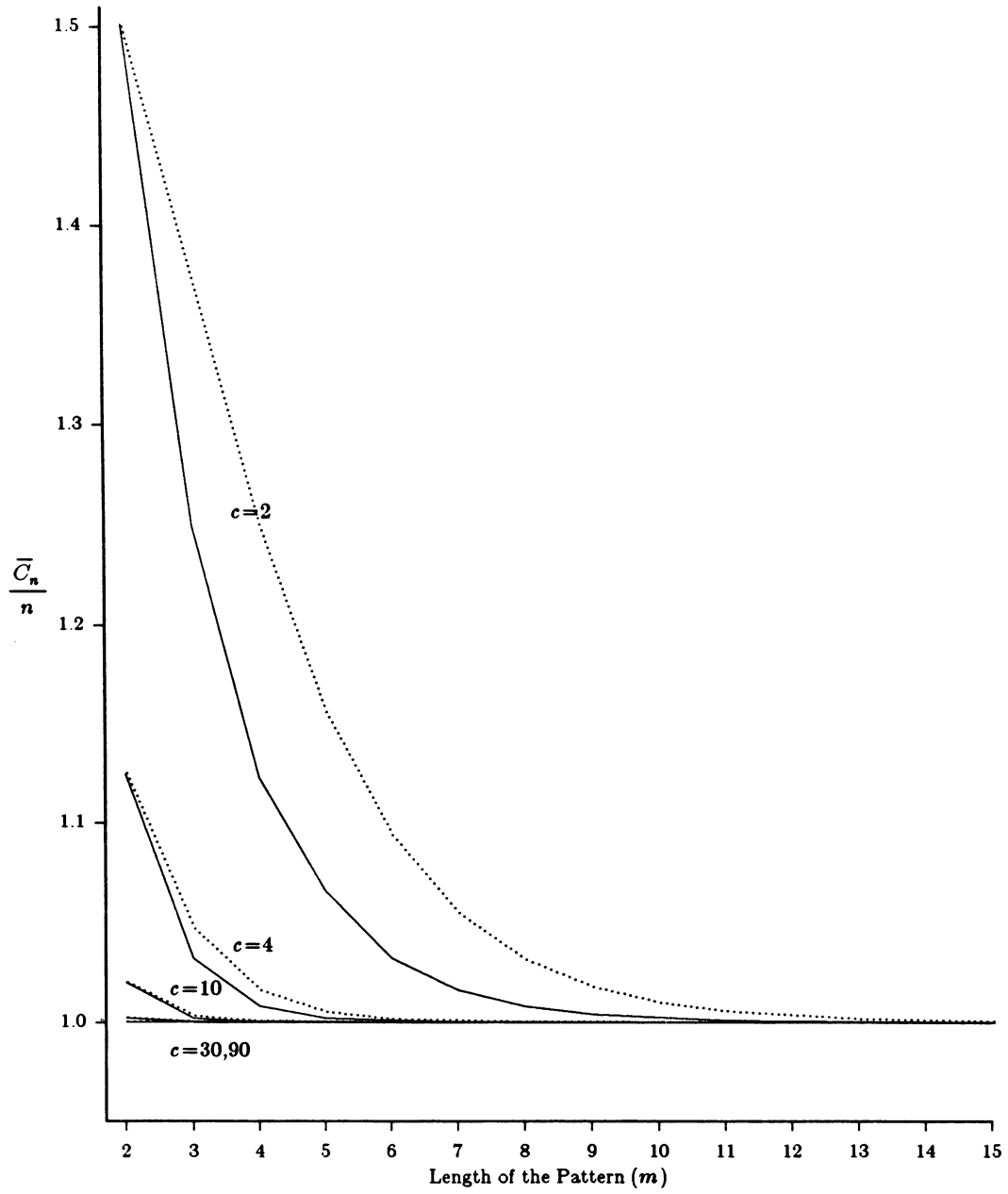


Figure 3. Simulations results for the Rabin-Karp algorithm  
in random text (dotted = theoretical results)



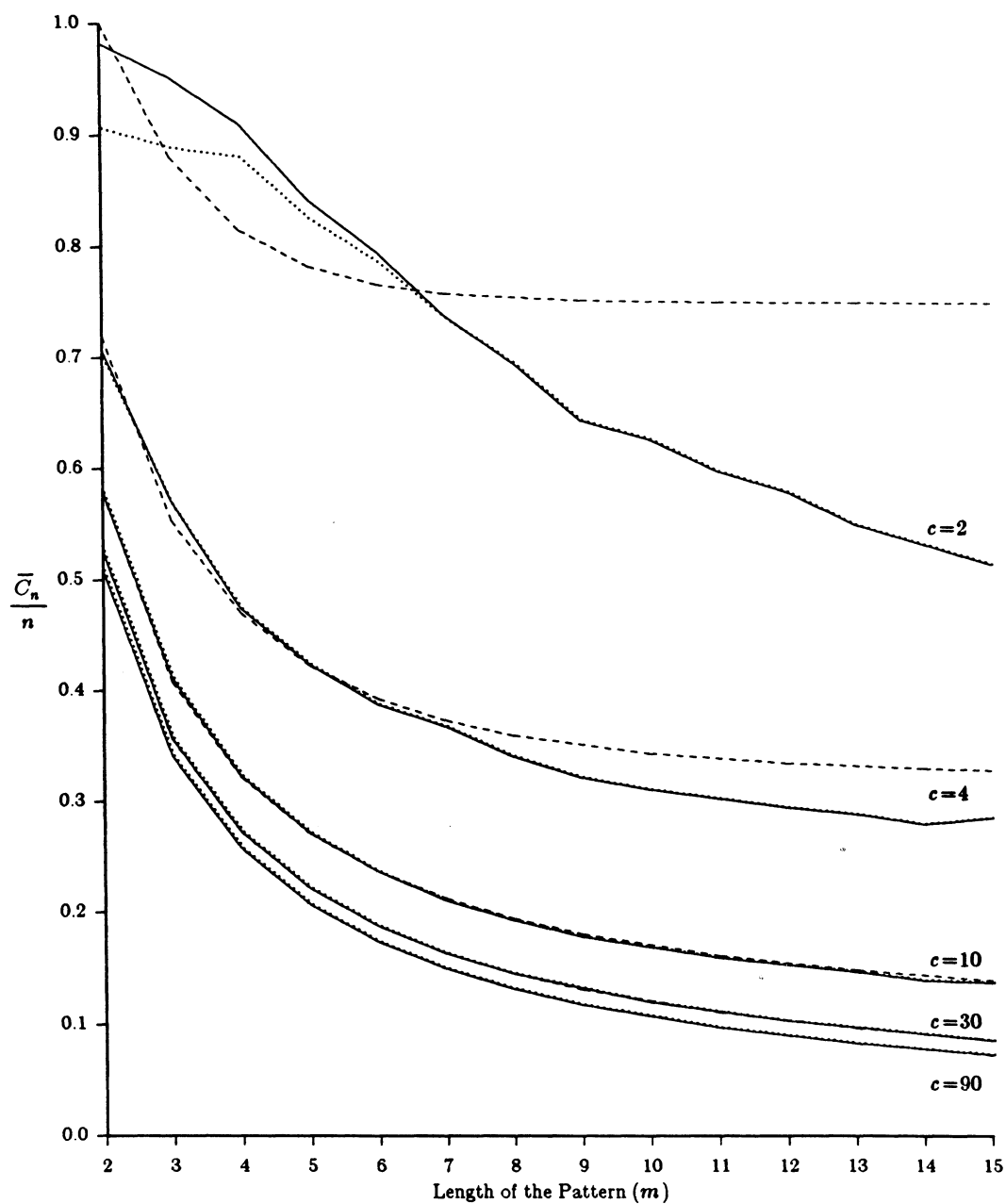


Figure 4. Simulations results for the Boyer-Moore algorithm in random text (dotted line = with Galil's improvement, dashed line=theoretical results)

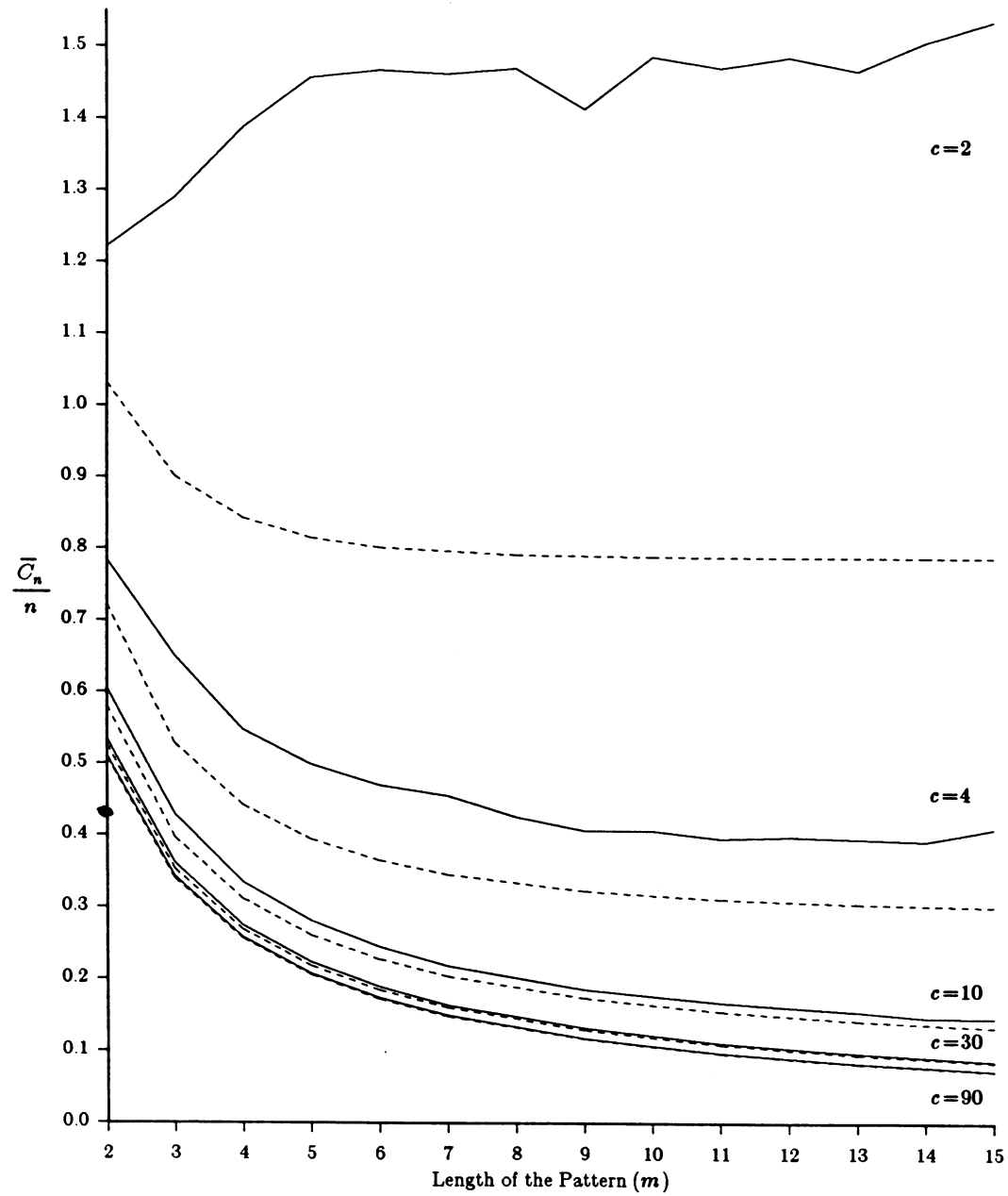


Figure 5. Simulations results for the Simplified Boyer-Moore algorithm in random text (dashed line = theoretical results)

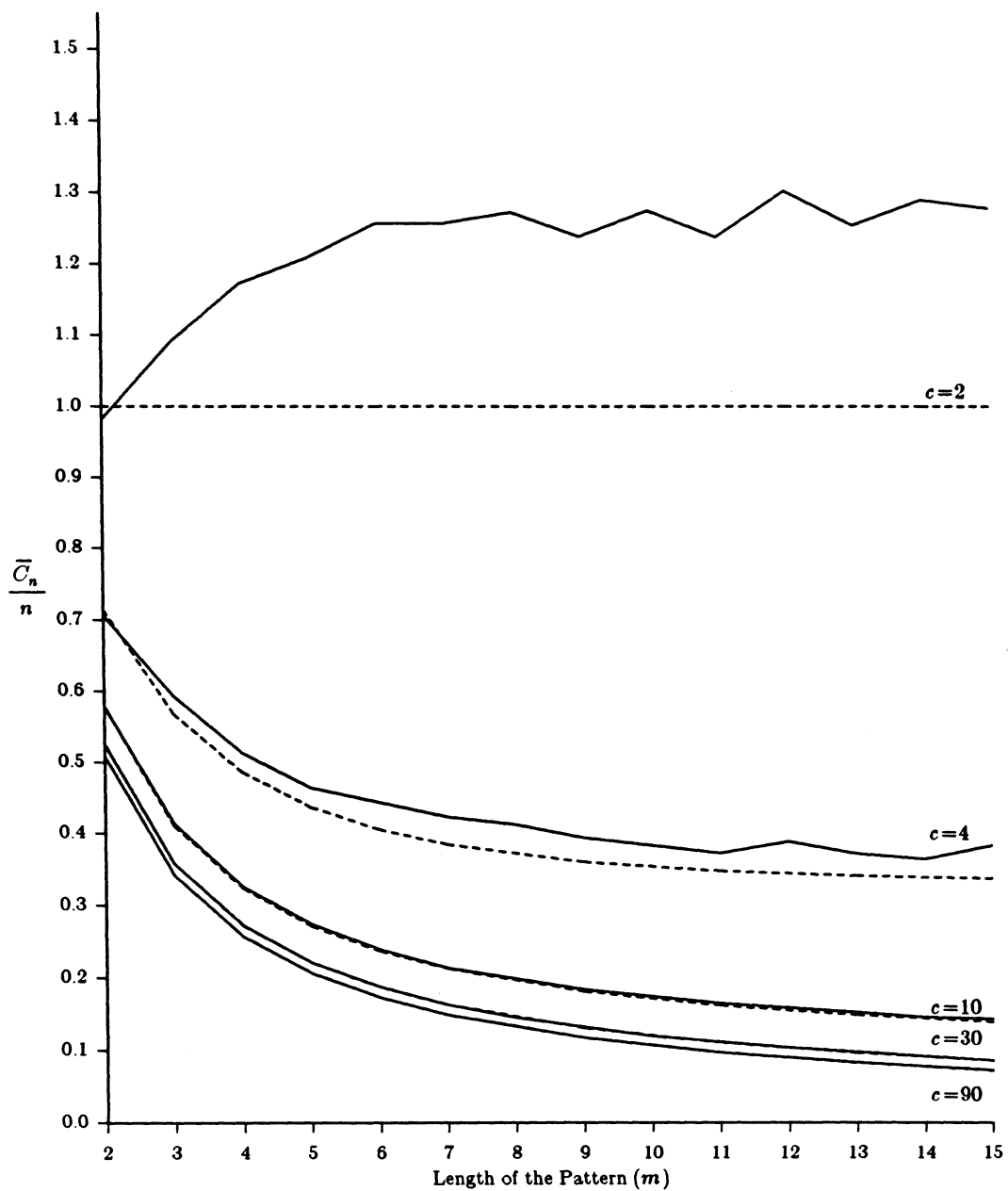


Figure 6. Simulation results for the Horspool version of the Boyer-Moore algorithm (dashed line=theoretical results)

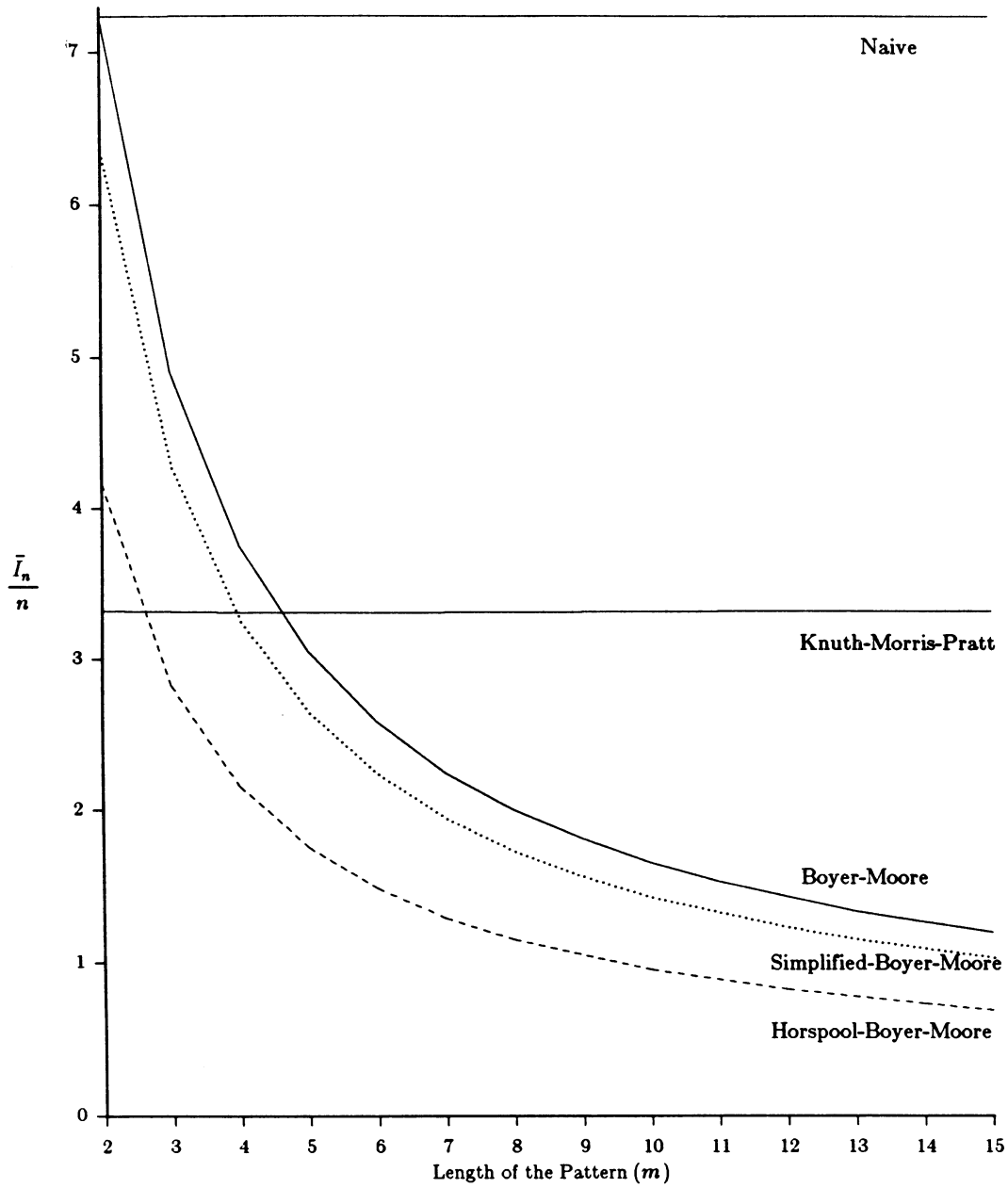


Fig 7. Simulations results for all the algorithms in random text ( $c=30$ )

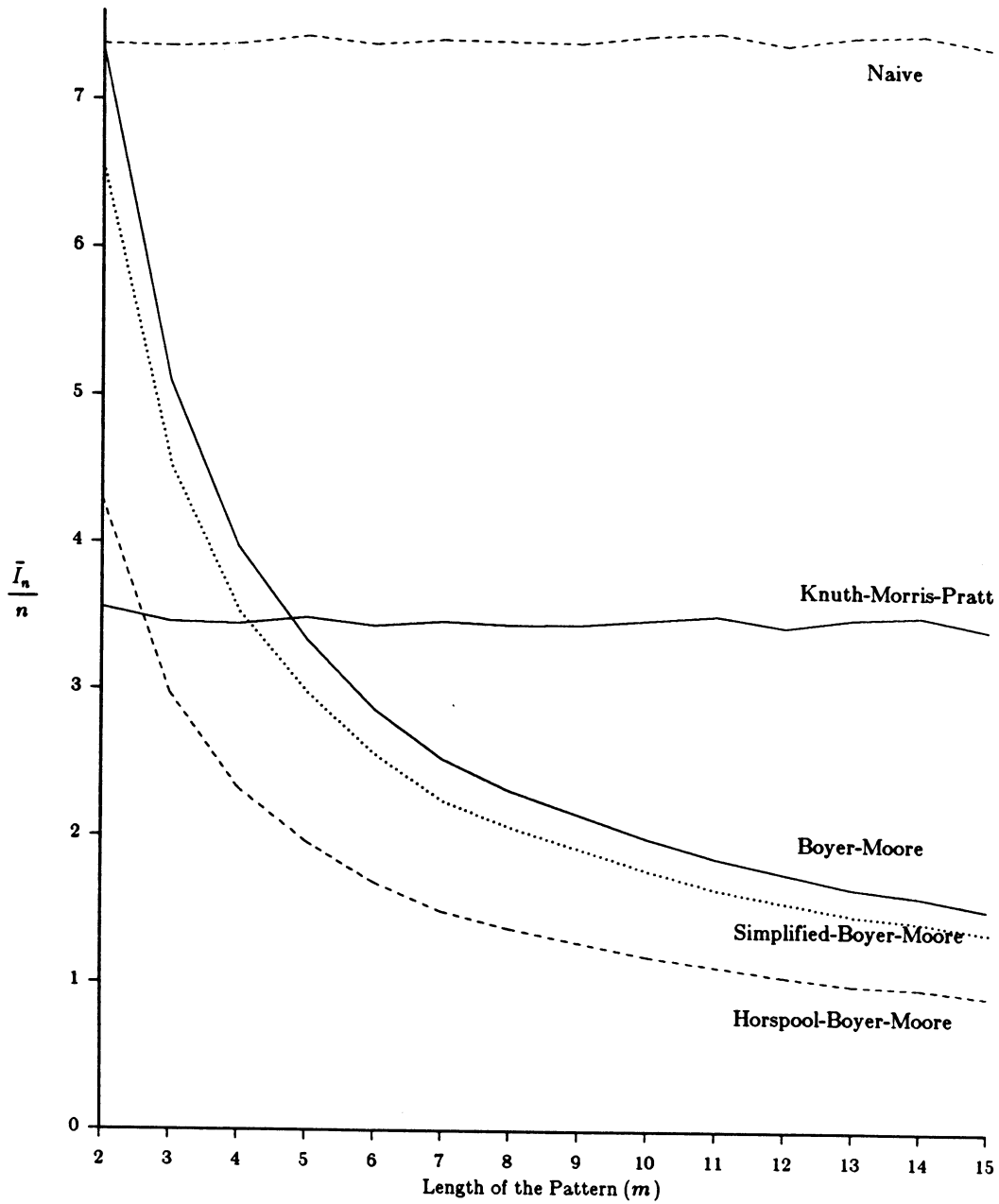


Fig 8. Simulations results for all the algorithms in real text

