Algorithms for String Searching: A Survey

Ricardo A. Baeza-Yates

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

Abstract

We survey several algorithms for searching a string in a piece of text. We include theoretical and empirical results, as well as the actual code of each algorithm. An extensive bibliography is also included.

1 Introduction

String searching is an important component of many problems, including text editing, data retrieval and symbol manipulation. The string searching or 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. 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.

We present the most important algorithms for string matching: the naive algorithm, the Knuth-Morris-Pratt algorithm [KMP77], different variants of the Boyer-Moore [BM77] algorithm, the shift-or algorithm [BYG89], and the Karp-Rabin [KR87] algorithm (a probabilistic one). Experimental results for random text and one sample of English text are included. We also survey the main theoretical results for each algorithm.

We use the C programming language [KR78] to present our algorithms.

[*]This work was supported by the Institute of Computer Research of the University of Waterloo and by the University of Chile.

2 Preliminaries

We use the following notation:

- n: The length of the text.
- m: The length of the pattern (string).
- c: The size of the alphabet Σ.
- $\overline{C}_n$: The expected number of comparisons performed by an algorithm while searching the pattern in a text of length n.

Theoretical results are given for the worst case number of comparisons, and 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 finite 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.

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. Alphabets of size $c = 4$ (DNA sequences) and $c = 30$ (approximately the number of lowercase English letters) 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 used. 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].

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, the probability of two characters being equal is $1/c$. Our random text model is similar to the one used in Knuth *et al* [KMP77] and Schaback [Sch88].

For example, the probability of finding a match between a random text of length m and a random pattern of length m is

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

and the expected number of matches of a random pattern of length m in a random text of length n is

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

3 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 1).

```
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 1: The naive or brute force string matching algorithm.

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.

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 [BY89c]

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

This is drastically different from the worst case mn.

4 The Knuth-Morris-Pratt Algorithm

The classic Knuth, Morris and Pratt algorithm, published in 1977 [MP70, 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

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

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 2.

Example 1: The *next* table for the pattern *abracadabra* is

	a	b	r	a	c	a	d	a	b	r	a	
next[j]	0	1	1	0	2	0	2	0	1	1	0	5

When the value in the next table is zero, we have to advance one position in the text and start comparing again from the beginning of the pattern. The last value of the next table (five) is used to continue the search after a match has been found. □

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 3).

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 algorithm. Variations that compute the *next* table on the fly are presented by Barth [Bar81] and Takaoka [Tak86]. Variations for the Aho and Corasick algorithm are presented in [BD80, Mey85].

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 expected number of comparisons for the KMP algorithm is bounded from above by [BY89c]

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

```
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 {                         /* Search */
        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 2: The Knuth-Morris-Pratt algorithm.

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 $\phi = (1+\sqrt{5})/2 \approx 1.618$.

A different analysis is presented by Regnier [R89]. 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.

```
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 3: Pattern preprocessing in the Knuth-Morris-Pratt algorithm.

Barth [Bar84] using a heuristic model, derived an approximation for the Knuth-Morris-Pratt algorithm, given by

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

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

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

Barth's approximation of Knuth-Morris-Pratt algorithm is too optimistic as shown by other theoretical and empirical results. 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 [Ŕ89], which implies, for $m > 2$,

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

5 The Boyer-Moore Algorithm

Also in 1977, 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.

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

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

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

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

Example 2: The $\widehat{dd}$ table for the pattern *abracadabra* is

```
             a  b  r  a  c  a  d  a  b  r  a
ddhat[j]    17 16 15 14 13 12 11 13 12  4  1
```

□

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

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

for every symbol x in the alphabet. See Figure 4 for the code to compute both tables ($\widehat{dd}$ and d) from the pattern.

Example 3: The d table for the pattern *abracadabra* is

```
d['a'] = 0   d['b'] = 2   d['c'] = 6   d['d'] = 4   d['r'] = 1
```

and the value for any other character is 11. □

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,

```
bmsearch( text, n, pat, m )   /* Search pat[1..m] in text[1..n] */
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 )        /* Search */
    {
        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: The Boyer-Moore algorithm.

their algorithm is incorrect. The corrected version can be found on Rytter's paper [Ryt80]; see Figure 5.

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]. Some experiments in a distributed environment are presented in [MNS84]. A variant of the BM algorithm when m is similar to n is given in [IA80]. A Boyer-Moore type algorithm to search a set of strings is presented in [CW79].

For a large alphabet of size c and $m \ll n$, we have [BY89b]

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

A different result, 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

```
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, dd )  /* 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's 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 5: Preprocessing of the pattern in the Boyer-Moore algorithm.

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.

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 proper 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, 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.

5.1 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 length of the overlapping characters. Of course the worst case is now $O(mn)$, but it will be faster on the average.

The analysis of this algorithm is presented in [BY89b]. 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 is asymptotically for n and m (with $m \ll n$)

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

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

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

5.2 The Boyer-Moore-Horspool Algorithm

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.

Horspool noted 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

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

Example 4: The d table for the pattern *abracadabra* is

```
d['a'] = 3   d['b'] = 2   d['c'] = 6   d['d'] = 4   d['r'] = 1
```

and the value for any other character is 11. □

The code for an efficient version of the Boyer-Moore-Horspool algorithm is extremely simple and is presented in Figure 6 where `MAX_ALPHABET_SIZE` is the size of the alphabet.

The expected value of a shift of the pattern over the text for the BMH algorithm is [BY89a]

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

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 ($n \ge m$) is [BY89c] asymptotically for n and m (with $m \ll n$)

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

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

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

Based on the theoretical 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. Improvements to the BMH algorithm for searching in English text are discussed in [BY89a, BY89b]. A hybrid algorithm that combines the BMH and KMP algorithms is proposed in [BY89c].

Figure 7 shows the expected number of comparisons per character for random text and $c = 4$. The code used is the one given in this paper, except the Knuth-Morris-Pratt algorithm that was implemented as suggested by their authors [KMP77] (the version given here is slower).

```
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 6: The Boyer-Moore-Horspool algorithm.

6 The Shift-Or Algorithm

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 [BYG89].

The main properties of the shift-or algorithm 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 algorithm is suitable for hardware implementation.

This algorithm is based on finite automata theory, as the KMP algorithm, and also exploits the finiteness of the alphabet, as in the BM algorithm.

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

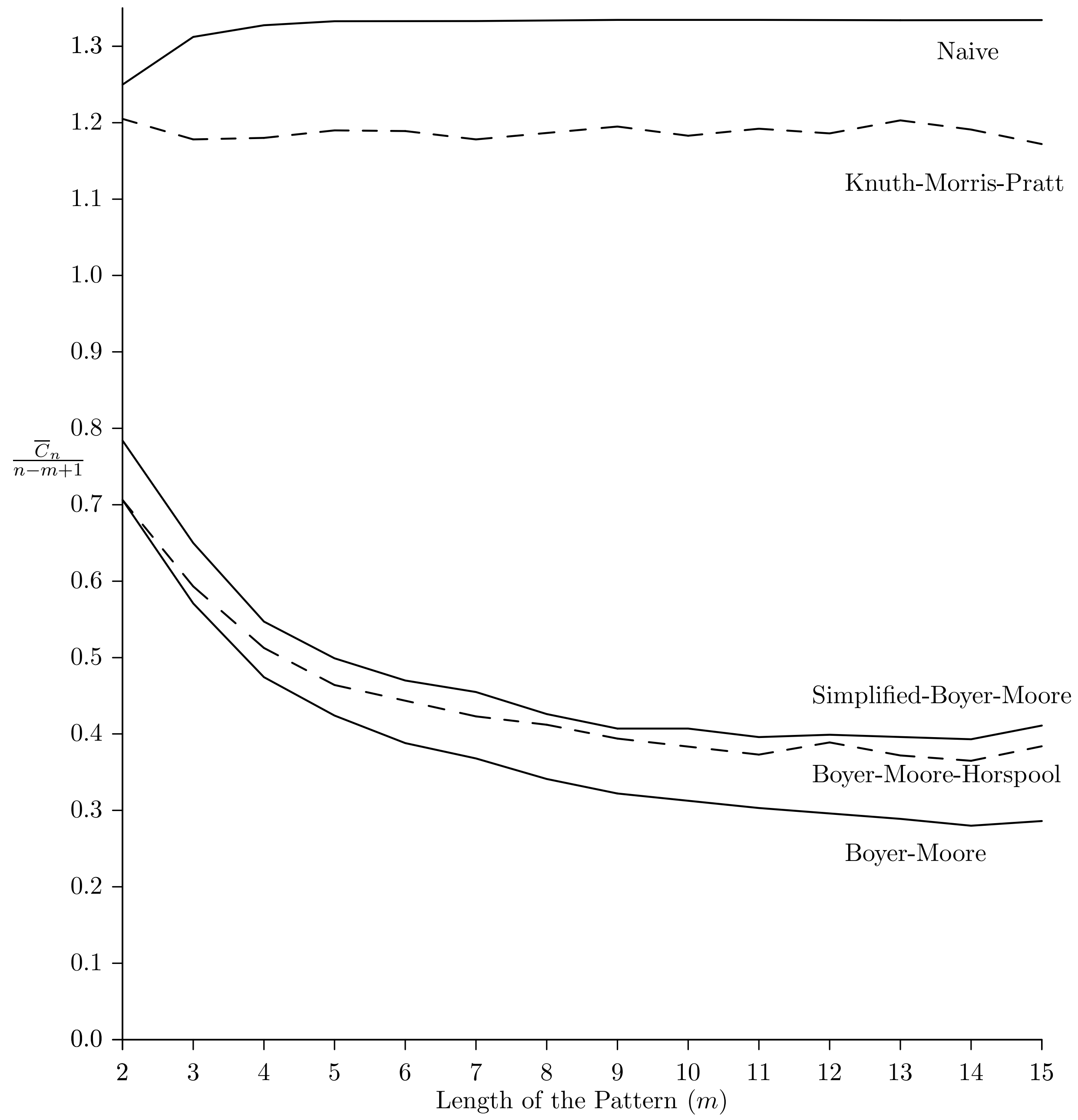

Figure 7: Expected number of comparisons for random text ($c = 4$).

$1, \ldots, i$ of the pattern and positions $(j-i+1), \ldots, j$ of the text, where j is the current position in the text.

We use 1 bit to represent each individual state, where the state is 0 if the last i characters have matched or 1 if not. Then, we can represent the vector state efficiently as a number in base 2 by:

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

where the s_i are the individual states. We have to report a match if s_m is 0, or equivalently if $state < 2^{m-1}$. That match ends at the current position.

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

- shift the vector state 1 bit 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 the bitwise operator *or* that, given the old vector state and the table value, gives the new state.

Then, each search step is:

$$state = (state\ \ <<\ \ 1)\ \ or\ \ T[curr\ char] \ ,$$

where << denotes the shift left operation.

The definition of the table T is

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

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

Example 5: 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
```

We finish the example, by searching for the first occurrence of *ababc* in the text *abdabababc*. 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 $O(\lceil \frac{m}{w} \rceil n)$, where $\lceil \frac{m}{w} \rceil$ is the time to compute a shift or other simple operation on numbers of m 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.

Figure 8 shows an efficient implementation of this algorithm. 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, that is, 1.

```
sosearch( text, n, pat, m )  /* Search pat[1..m] in text[1..n] */
register char *text;
char pat[];
int n, m;
{
    register char *end;
    register unsigned int state, lim;
    unsigned int T[MAXSYM], i, j;
    char *start;

    if( m > WORD )
        Abort( "Use pat size <= word size" );
    for( i=0; i<MAXSYM; i++ ) T[i] = ~0;        /* Preprocessing */
    for( lim=0, j=1, i=1; i<=m; lim |= j, j <<= B, i++ )
        T[pat[i]] &= ~j;
    lim = ~(lim >> B);
    text++; end = text+n+1;                     /* Search */
    state = ~0;                                 /* Initial state */
    for( start=text; text < end; text++ )
    {
        state = (state << B) | T[*text]; /* Next state */
        if( state < lim ) Report_match_at_position( text-start-m+2 );
    }
}
```

Figure 8: Shift-Or algorithm for string matching (simpler version).

The changes needed for a more efficient implementation (that is, scan the text until we see the first character of the pattern [KMP77]) of the algorithm are shown in Figure 9. (using structured programming!). The speed of this version depends on the frequency of the first letter of the pattern in the text. The empirical results for this code are shown in Figures 11 and 12. Another implementation is possible using the bitwise operator *and* instead of the *or* operation, and complementing the value of T_x for all $x \in \Sigma$.

7 The Karp-Rabin Algorithm

A different approach to string searching is to use hashing techniques [Har71]. 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.

```
initial = ~0; first = pat[1]; start = text; /* Search */
do {
    state = initial;
    do {
        state = (state << B) | T[*text]; /* Next state */
        if( state < lim ) Report_match_at_position( text-start-m+2 );
        text++;
    } while( state != initial );
    while( text < end && *text != first ) /* Scan */
        text++;
} while( text < end );
```

Figure 9: Shift-Or algorithm for string matching.
(Version based on implementation of [KMP77].)

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 10 (`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

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

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 $O(1/n)$.

```
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 10: The Karp-Rabin algorithm.

However, in practice, the bound M depends on the word size used for the arithmetic operations, and not on m or n. Based on a more realistic model of computation, we have the following results [GBY90].

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.

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.

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

The empirical results are compared with our theoretical results in Table 1 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 [Gon88]. 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 10) 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.

8 Conclusions

We have presented the most important string searching algorithms. Figure 11 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 12 shows the same empirical results of Figure 11, but for English text instead of random text. The results are similar. For the shift-or algorithm, the given results are for the efficient version. The results for the Karp-Rabin algorithm are not included, because in all cases the time is bigger than 300 seconds.

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,

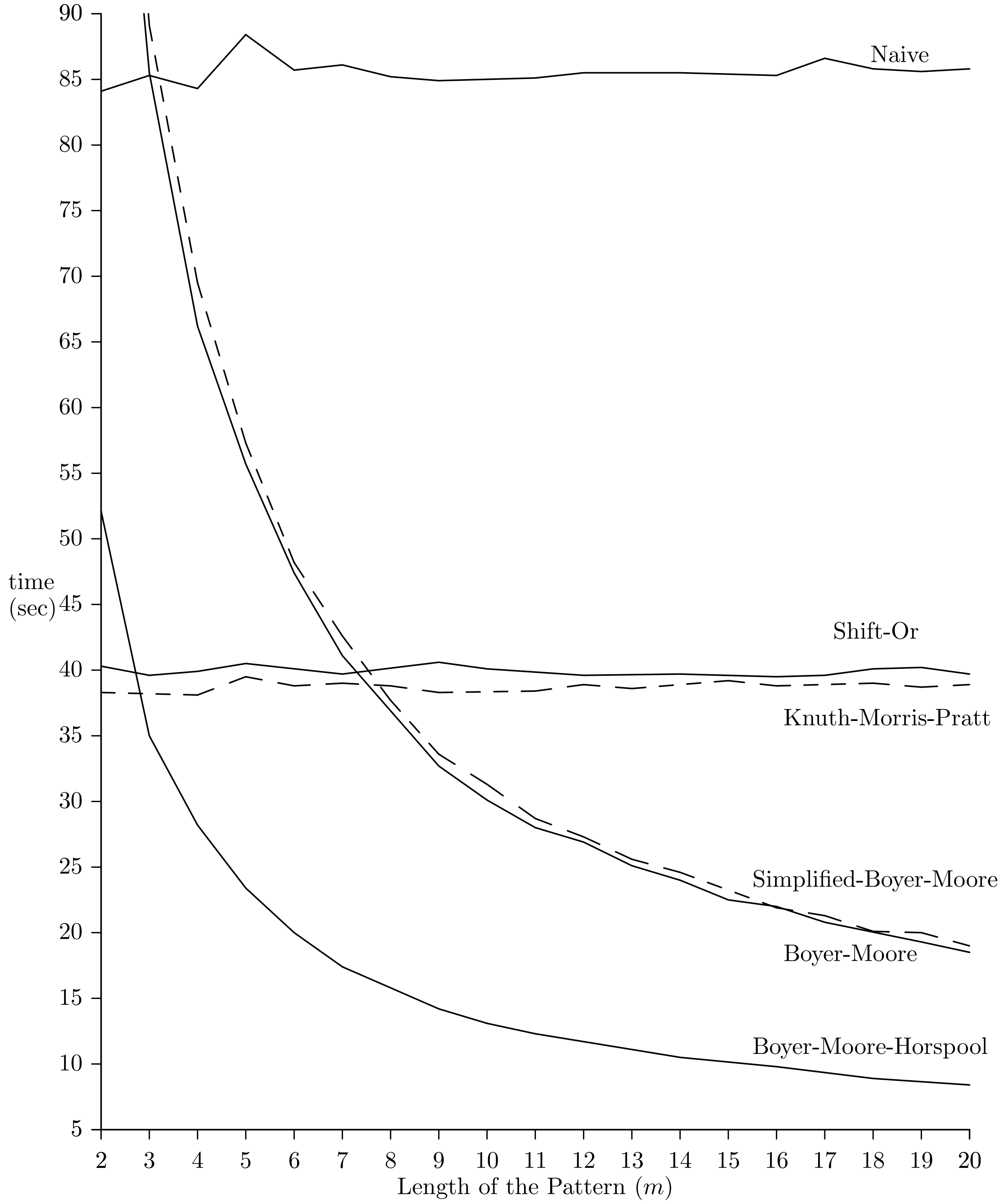

Figure 11: Simulation results for all the algorithms in random text ($c = 30$).

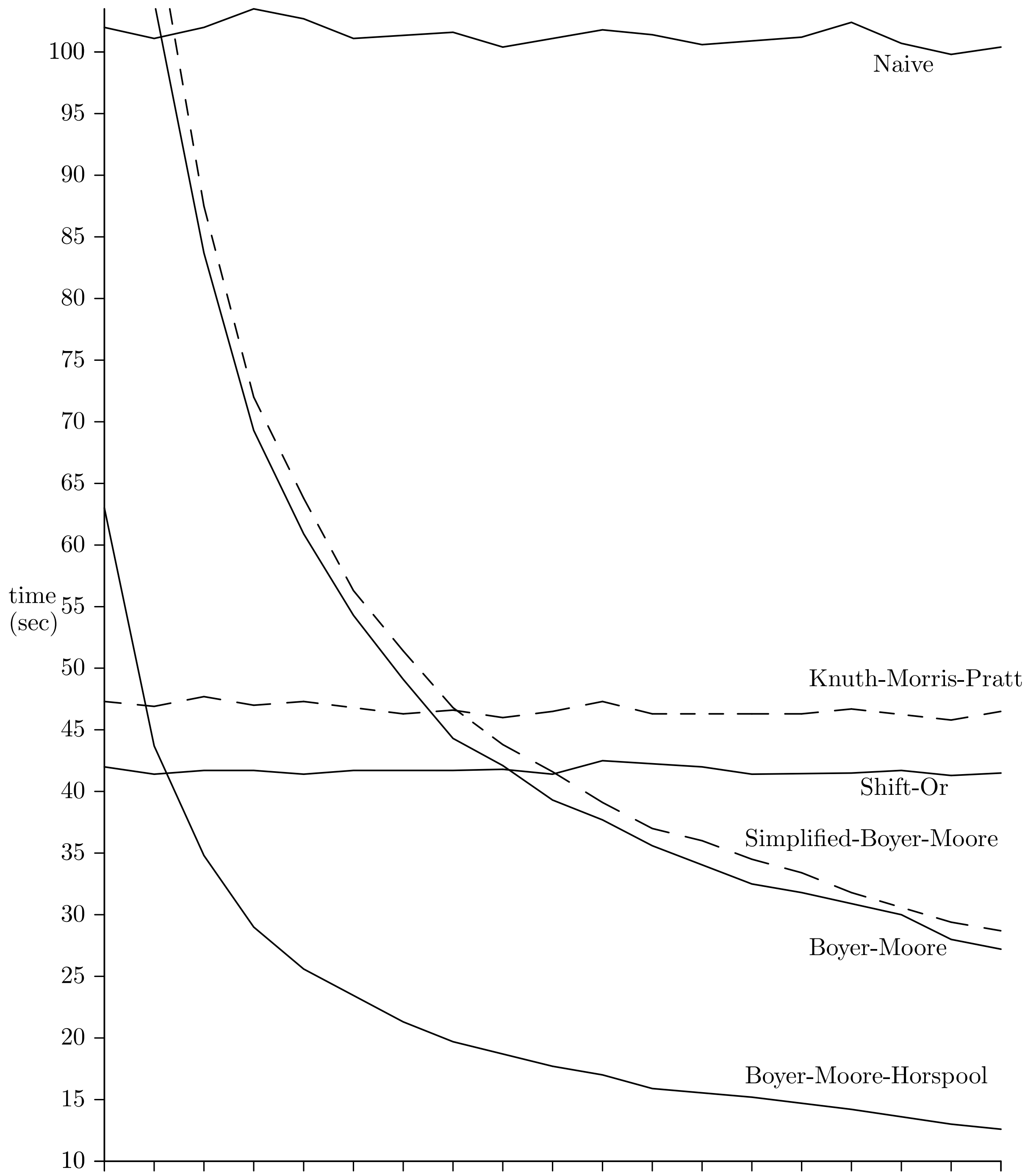

Figure 12: Simulation results for all the algorithms in English text.

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.

The shift-or algorithm has a running time similar to the KMP algorithm. However, the main advantage of this algorithm, is that we can search for more general patterns ("don't care" symbols, complement of a character, etc.) using exactly the same searching time [BYG89] (only the preprocessing is different).

None of the algorithms surveyed is optimal on the average number of comparisons. In Knuth-Morris-Pratt [KMP77], it is shown that there exists an algorithm for which the expected worst case number of inspected characters is $O(\frac{\log_c m}{m} n)$ in random text, where c is the size of the alphabet. In 1979, Yao [Yao79] proved that this result was optimal, showing that the average number of characters that need to be inspected in a random string of length n is

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

for almost all patterns of length m ($n > 2m$). This lower bound also holds for the "best case" complexity. A theoretical algorithm with this average search time is also presented (see also [Sch88, Li84, LY86]).

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 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 (see also [Sli80, Sli83]). Also they show that the delay between reading two characters of the text is bounded by a constant (this is interesting for real time searching algorithms) [Gal81]. Practical algorithms that achieve optimal worst case time and space are presented in [CP88, Cro88, CP89]. Optimal parallel algorithms for string matching are presented by Galil and by Vishkin [Gal85, Vis85] (see also [BBG$^+$89, KLP89]).

Many of the algorithms presented may be implemented with hardware [Has80, Hol79]. For example, Aho and Corasick machines [AYS85, CF87, WKY85].

If we allow preprocessing of the text, we can search a string in worst case time proportional to its length. This is achieved by using a Patricia tree [Mor68] as index. This solution needs $O(n)$ extra space and preprocessing time, where n is the size of the text [Wei73, McC76, Mor68, MR80, KBG87]. For other kind of indices for text see [Fal85, Gon83, BBH$^+$85, BBE$^+$87].

Acknowledgements

We wish to acknowledge the helpful comments of Gaston Gonnet, Frank Tompa, and Derick Wood.

References

[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.

[AYS85] J. Aoe, Y. Yamamoto, and R. Shimada. An efficient implementation of static string pattern matching machines. In *IEEE Int. Conf. on Supercomputing Systems*, volume 1, pages 491–498, St. Petersburg, Fla, 1985.

[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.

[BBE$^+$87] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell. Completed inverted files for efficient text retrieval and analysis. *J.ACM*, 34:578–595, 1987.

[BBG$^+$89] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parellelizable problems. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 309–319, Seattle, Washington, 1989.

[BBH$^+$85] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[BD80] T.A. Bailey and R.G. Dromey. Fast string searching by finding subkeys in subtext. *Inf. Proc. Letters*, 11:130–133, 1980.

[BM77] R. Boyer and S. Moore. A fast string searching algorithm. *C.ACM*, 20:762–772, 1977.

[BY89a] R. Baeza-Yates. Improved string searching. *Software-Practice and Experience*, 19(3):257–271, 1989.

[BY89b] R.A. Baeza-Yates. *Efficient Text Searching.* PhD thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.

[BY89c] R.A. Baeza-Yates. String searching algorithms revisited. In F. Dehne, J.-R. Sack, and N. Santoro, editors, *Workshop in Algorithms and Data Structures*, pages 75–96, Ottawa, Canada, August 1989. Springer Verlag Lecture Notes on Computer Science 382.

[BYG89] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. In *Proc. of 12th ACM SIGIR*, pages 168–175, Cambridge, Mass., June 1989. (Addendum in ACM SIGIR Forum, V. 23, Numbers 3, 4, 1989, page 7.).

[CF87] H. Cheng and K. Fu. VLSI architectures for string matching and pattern matching. *Pattern Recognition*, 20:125–141, 1987.

[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.

[Fal85] C. Faloutsos. Access methods for text. *ACM C. Surveys*, 17:49–74, 1985.

[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. Optimal parallel algorithms for string matching. *Information and Control*, 67:144–157, 1985.

[GBY90] G.H. Gonnet and R.A. Baeza-Yates. An analysis of the Karp-Rabin string matching algorithm. *Information Processing Letters*, 34:271–274, 1990.

[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.

[Gon88] G.H. Gonnet. Private communication, 1988.

[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.

[Har71] M.C. Harrison. Implementation of the substring test by hashing. *C.ACM*, 14:777–779, 1971.

[Has80] R. Haskin. Hardware for searching very large text databases. In *Workshop Computer Architecture for Non-Numeric Processing*, volume 5, pages 49–56, California, 1980.

[Hol79] L.A. Hollaar. Text retrieval computers. *IEEE Computer*, 12:40–50, 1979.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, 10:501–506, 1980.

[IA80] S. Iyengar and V. Alia. A string search algorithm. *Appl. Math. Comput.*, 6:123–131, 1980.

[KBG87] M. Kemp, R. Bayer, and U. Guntzer. Time optimal left to right construction of position trees. *Acta Informatica*, 24:461–474, 1987.

[KLP89] Z. Kedem, G. Landau, and K. Palem. Optimal parallel suffix-prefix matching algorithm and applications. In *SPAA'89*, Santa Fe, New Mexico, 1989.

[KMP77] D.E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J on Computing*, 6:323–350, 1977.

[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.

[Li84] M. Li. Lower bounds on string-matching. Technical Report TR-84-636, Dept. of Computer Science, Cornell University, 1984.

[LY86] M. Li and Y. Yesha. String matching cannot be done by a two-head one way deterministic finite automaton. *Inf. Proc. Letters*, 22:231–235, 1986.

[McC76] E. McCreight. A space-economical suffix tree construction algorithm. *J.ACM*, 23:262–272, 1976.

[Mey85] B. Meyer. Incremental string matching. *Inf. Proc. Letters*, 21:219–227, 1985.

[MNS84] P. Moller-Nielsen and J. Staunstrup. Experiments with a fast string searching algorithm. *Inf. Proc. Letters*, 18:129–135, 1984.

[Mor68] D. Morrison. PATRICIA-Practical algorithm to retrieve information coded in alphanumeric. *JACM*, 15:514–534, 1968.

[MP70] J.H. Morris and V.R. Pratt. A linear pattern matching algorithm. Technical Report 40, Computing Center, University of California, Berkeley, 1970.

[MR80] M. Majster and A. Reiser. Efficient on-line construction and correction of position trees. *SIAM J on Computing*, 9:785–807, 1980.

[R8́9] M. Régnier. Knuth-Morris-Pratt algorithm: An analysis. In *MFCS'89, Lecture Notes in Computer Science 379*, pages 431–444, Porabka, Poland, August 1989. Springer-Verlag. Also as INRIA Report 966, 1989.

[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.

[Sli80] A. Slisenko. Determination in real time of all the periodicities in a word. *Sov. Math. Dokl.*, 21:392–395, 1980.

[Sli83] A. Slisenko. Detection of periodicities and string-matching in real time. *Journal of Soviet Mathematics*, 22:1316–1386, 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.

[Vis85] U. Vishkin. Optimal parallel pattern matching in strings. *Information and Control*, 67:91–113, 1985.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *FOCS*, volume 14, pages 1–11, 1973.

[WKY85] S. Wakabayashi, T. Kikuno, and N. Yoshida. Design of hardware algorithms by recurrence relations. *Systems and Computers in Japan*, 8:10–17, 1985.

[Yao79] A.C. Yao. The complexity of pattern matching for a random string. *SIAM J on Computing*, 8:368–387, 1979.