Ricardo Baeza-Yates and Gaston H. Gonnet

A New Approach to Text Searching

String searching is a very important component of many problems, including text editing, bibliographic retrieval, and symbol manipulation. Recent surveys of string searching can be found in [4, 18].

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

The main idea is to represent the state of the search as a number, and each search step does 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 $m \leq w$, being w the word size in bits of the computer used, we have an $O(n)$ time algorithm using $O(|\Sigma|)$ extra space and $O(m + |\Sigma|)$ preprocessing time, where Σ denotes the alphabet.

For string matching, empirical results show that the new algorithm compares favorably with the Knuth-Morris-Pratt (KMP) algorithm [24] for any pattern length and the Boyer-Moore (BM) algorithm [12] for short patterns (up to length 6). For patterns with *don't care* symbols and complement symbols, this is the first practical and efficient algorithm in the literature, and it can be generalized to any finite class of symbols or their complements. For searches with at most k mismatches, this algorithm is three times faster than any known algorithm for $m < 9$.

The main properties of this class of algorithms are:

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

It is worth noting that the BM algorithm needs to buffer the text.

All these properties indicate that this class of algorithm is suitable for hardware implementation. For these reasons, we believe this new approach is a valuable contribution to all applications dealing with text searching. A preliminary version of this article was presented in [9].

The Shift-Add Approach to String Matching

Our algorithm is based on finite automata theory, as in the Knuth-Morris-Pratt algorithm [24], and also exploits the finiteness of the alphabet, as in the Boyer-Moore algorithm [12].

Let *pat* be a pattern of length m, and *text* a text of length n. 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 tells us the state of the search between the positions $1, \ldots, i$ of the pattern and positions $(j - i + 1), \ldots, j$ of the text, where j is the current position in the text. Intuitively, we can

think we have m string comparators running in parallel and reading concurrently the same text position. This analogy, called bit-parallelism, is used to introduce our approach in a recent survey [5].

Let s_i^j be the set of states (for $1 \leq i \leq m$) after reading the j-th character of the text. Namely, s_i^j is the number of mismatches between $pat_1, \ldots, pat_i$ and $text_{j-i+1}, \ldots, text_j$ (characters that are different in the corresponding positions). Then, if $s_m^j = 0$ we have found a match ending at the current position in the text.

Let *ababc* be the pattern ($m = 5$). Suppose we are in position $j - 1$ in the text given in Figure 1(a) searching this pattern. Figure 1(a) gives the value of s_i^{j-1}. If we advance one position in the text, we have a new value for s_i^j (Figure 1(b)). Let $T[x]$ be a table such that

$$T_i[x] = \begin{cases} 0 & x = pat_i \\ 1 & \text{otherwise} \end{cases}$$

The value for $T[a]$ is given in Figure 1. It is not difficult to see that

$$s_i^j = s_{i-1}^{j-1} + T_i[text_j],$$

defining $s_0^j = 0$ for all j. For this reason, we call this algorithm the **shift-add** algorithm.

Assume we have arbitrary precision arithmetic using a word of size w, and the dependency of the current position j is implicit. Suppose we need b bits to represent each individual state s_i, where as we shall see later, b depends on the searching problem. Then, we can represent the vector state efficiently as a number in base 2^b by:

$$state_j = \sum_{i=0}^{m-1} s_{i+1}^j 2^{b \cdot i}.$$

For string matching we need only 1 bit (that is $b = 1$), where s_i is 0 if the last i characters have matched, or 1 if they have not. That is,

$$s_i^j = \{pat_1, \ldots, pat_i \neq ? text_{j-i+1}, \ldots, text_j\}$$

We have to report a match if s_m^j is 0, or equivalently if $state_j < 2^{m-1}$.

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

- shift the vector state b bits to the left to reflect that we have advanced one position in the text. In practice, this sets s_0^j to be 0 by default.
- update the individual states according to the new character. For this, we use the table T that is defined by preprocessing the pattern with one entry per alphabet symbol, and an operator *op* that, given the old vector state and the table value, gives the new state. Note that this works only if the operator in the individual state s_i^j does not produce a carry that affects state s_{i+1}^j.

Then, each search step changes the state using the assignment:

$$state_j = (state_{j-1} << b) \; op \; T[text_j],$$

where $<<$ denotes the bitwise shift-left operation.

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

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

for every symbol x of the alphabet Σ, where $\delta(C)$ is 1 if the condition C is true, and 0 otherwise. An implementation is presented in a subsequent section of this article.

We need $b \cdot m \cdot |\Sigma|$ bits of extra memory, and if the word size w is at least $b \cdot m$, only $|\Sigma|$ words are needed. We set up the table T by preprocessing the pattern before the search. This can be done in $O\left(\left\lceil \frac{mb}{w} \right\rceil (m + |\Sigma|)\right)$ time.

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

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

A choice for *op* for the case of exact string matching is a bitwise logical *or*. We finish the example by searching for the first occurrence of *ababc* in the text *abdabababc*. The initial state is 11111. Table 1 provides an example of this searching phase. 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 $O\left(\left\lceil \frac{mb}{w} \right\rceil n\right)$, where $\left\lceil \frac{mb}{w} \right\rceil$ is the time to compute a constant number of operations on integers of mb bits using a word size of w bits. In practice (patterns of length up to the word size: 32 or 64 bits) we have $O(n)$ worst- and average-case time. In the uniform cost RAM model, w is $O(\log_2 n)$ where n is the size of our problem (in this case, the size of the text*). The same applies to the size of the alphabet. So, we can say that our string-searching algorithm has $O(mn/\log n)$ time and $O(m)$ space complexity.

For each kind of pattern or searching problem, we can choose b and *op* appropriately (as in next sections). A similar idea was presented by Gonnet [17] applied to searching text using signatures.

String Matching with Classes

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

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

*In practice, with two 32-bit words we can address any conceivable text.

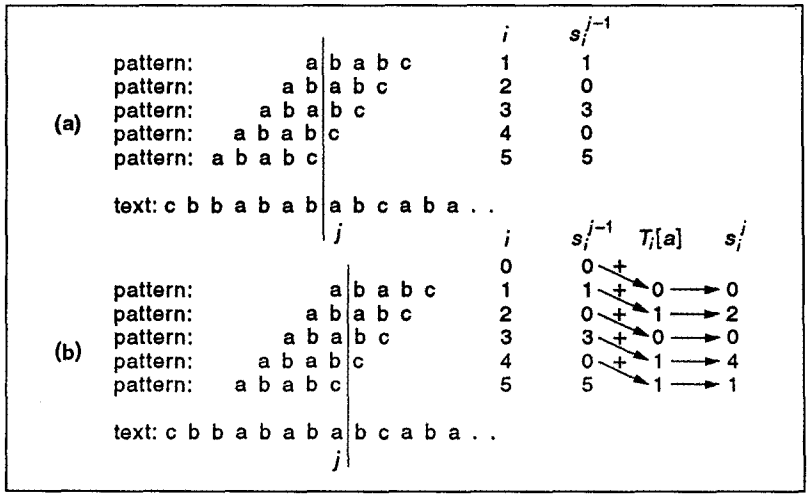


Figure 1. Example for the shift-add algorithm

For example, the pattern $[Pp]a\overline{[aeiou]}*\overline{a}[p..tv..z]$ matches the word *Patter,* but not *python* or *Patton.* Let m' be the size of the description of the pattern (that is, the number of elements in each class, with the size of * considered 1 and with complements not taken in account), and m the size of the pattern. For the previous example, $m' = 20$ and $m = 6$.

String matching with *don't care* symbols was addressed before by Fischer and Paterson [15] achieving

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

asymptotic search time, and also by Pinter [26] including complement symbols (same complexity). However, these are theoretical results, and their algorithms are not practical. Pinter also gives a $O(mn)$ algorithm that is faster than a naive algorithm. For small patterns, the complexity of our algorithm is much better, and is much easier to implement. A similar class of patterns is considered by Abrahamson [1], where a theoretical algorithm that runs in

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

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

Attempts to adapt the KMP algorithm to this case have failed [15, 26], and for the same reason the BM algorithm as presented in Knuth et al. [24] cannot solve this problem. It is possible to use the Horspool version of the BM algorithm [19], but the worst case is $O(mn)$; and on average, if we have a *don't care* character near the end of the pattern, the whole idea of the shift table is worthless. By mapping a class of characters to a unique character, the Karp and Rabin algorithm [21] solves this problem too. However, this is a probabilistic algorithm, and if we check each reported match, the search time is $O(n + m + mM)$, where M is the number of matches. Potentially, $M = O(n)$, and their algorithm is slow in practice, because of the use of multiplications and modulus operations.

To search for these extended patterns, we need only to modify the table T, such that, for each position, we process every character in the class. That is,

$$T[x] = \sum_{i=0}^{m-1} \delta(x \notin Class_{i+1}) 2^{b \cdot i},$$

where $Class_i$ is the class of characters for the ith position of the pattern.

Example 2: Let $\{a, b, c, d\}$ be the alphabet, and $a\overline{b}[ab]b\overline{[a..c]}$ the pattern. We have $m = 5$ and $m' = 8$. Then, if $b = 1$ (as for string matching), the entries for the table T are:

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

To maintain $O\left(\left\lceil \frac{mb}{w} \right\rceil (m' + |\Sigma|)\right)$ preprocessing time (instead of $O\left(\left\lceil \frac{mb}{w} \right\rceil m'|\Sigma|\right)$ time), where m' is the size of the description of the pattern, we initially represent $T[x]$ in the alphabet as 0 for bit positions corresponding to *don't care* symbols and complements. In the worst case $m' = O(m|\Sigma|)$; however in practical queries m' is similar to m. Figure 2 shows the preprocessing phase in pseudocode for this class of patterns, where the notation $111..0_i..111$ means a sequence of 1s with a 0 in the ith position. The search time remains the same.

We can extend the algorithm to Soundex-like classes [23], or to special symbols in the text. For example, if we have *don't care* symbols in the text, we define $T[*] = 0$.

Table 1. String searching example

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	

String Matching with Mismatches

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

Landau and Vishkin [25] give the first efficient algorithm to solve this particular problem. Their algorithm uses $O(k(n + m \log m))$ time and $O(k(n + m))$ space. While it is fast, the space required is unacceptable for practical purposes. Galil and Giancarlo [16] improve this algorithm to $O(kn + m \log m)$ time and $O(m)$ space. However, this algo-

rithm uses a static lowest common ancestor algorithm over a suffix tree. Thus, the constant involved in the linear term is large, being slower than the Landau and Vishkin algorithm on practice. Other approaches to this problem are presented in [6, 8].

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

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

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

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

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

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

bits. At each step we record the overflow bits in an overflow state, and we reset the overflow bits of all individual states (in fact, we only have to do this each k steps, but such improvement is not practical). The new search algorithm is shown in Figure 3.

Note that if $k > m/2$, we should count matches instead of mismatches (that is, $b = O(\log(m - k))$ if $k > m/2$). The only problem for this case is that it is not possible to tell how many errors there are in a match. Table 2 shows maximal values for m when constrained to using a 32-bit word.

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

Example 3: Let $\{a, b, c, d\}$ be the alphabet, and *ababc* the pattern (see Example 1 for the values of the table T). We want to search for all occurrences of *ababc* with at most 2 mismatches in the text *abdababababc*. Because the value of b is 3 for 2 mismatches, every position in the state is represented by a number in the range 0–4. The initial state is 00000 and the initial overflow is 44444. (See Table 3.) We report a match when the sum of the leftmost digits of the state and the overflow is less than 3. In this case, there is a match at position 4 with one mismatch (detected at position 8), and another match at position 6 with no mismatches (detected at position 10).

It is possible to use only $b = \lceil log_2(k + 1) \rceil$ bits by performing $O(\log \log k)$ operations in the loop. The idea is to add 1 except if each b-bit slice has only ones. To detect this we use shifts and bitwise-ands. For example, if $b = 5$, we perform the operations shown in Table 4, where x will have a 0 in the least significant bit of every 5-bit individual state if all bits in each 5-bit slice are 1s, or 1 otherwise. It is not difficult to show we need $O(\log \log k)$ operations. For example, if b is 8, we shift by 4, 2 and 1 bits. Table 1 also shows the maximum value of m for this case ($w = 32$).

If we have mismatches with different cost, we change the definition of T to reflect this. In this case, instead of a number of mismatches k, we have a maximum allowed cost. For example, we can define that a mismatch between vowels costs 1, and that other mismatches cost 2 (see also [8]).

Multiple Patterns

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

The KMP algorithm and the BM algorithm have already been extended to this case (see [2] and [13] respectively), achieving a worst-case time of $O(n + m)$, where m is the total length of the set of patterns.

If we have to search for $p_1 + \cdots + p_\ell$, and we keep one vector state per pattern, we have an immediate $O\left(\left\lceil \frac{m_{max}}{w} \right\rceil \ell n\right)$ time algorithm for a set of ℓ patterns, where $m_{max} = \max_i(|p_i|)$. However, we can coalesce all the vectors, keeping all the information in only one vector state

```
initval ← 111...111
for i from 1 to m {
    if pat_i is * or a complement then
        initval ← initval & 111..0_i..11
}
for i from 1 to |Σ| {
    T[x_i] ← initval
}
for i from 1 to m {
    for all x ∈ pat_i {
        if pat_i is a complement then
            T[x] ← T[x] | 000..1_i..00
        else
            T[x] ← T[x] & 111..0_i..11
    }
}
```

Figure 2. Pseudocode for preprocessing of pattern with classes

```
mask ← 1_mb0..01_2b0..01_b0..0
lim ← (k + 1) << (m − 1)b
for i from 1 to n {
    state ← (state << b) + T[text_i]
    overf ← (overf << b) | (state and mask)
    state ← state & mask
    if (state + overf) < lim then
        print (match at current position)
}
```

Figure 3. Improved Search algorithm for string matching with at most k mismatches

achieving $O\left(\left\lceil \frac{m_{sum}}{w} \right\rceil n\right)$ search time, with $m_{sum} = \sum_i |p_i|$. The disadvantage is that now we need numbers of size m_{sum} bits, and $O(|\Sigma| m_{sum})$ extra space.

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

Implementation

In this section we present efficient implementation of the various algorithms that count the number of matches of patterns in a text using single-word integers. Algorithms with different actions in case of a match are easily derived from them. We also include some experimental results.

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

- MAXSYM: Size of the alphabet. For example, 128 for ASCII code.
- WORD: Word size in bits (32 in our case).
- B: Number of bits per individual state (1 for string matching, variable for string matching with mismatches).
- EOS: End of string (0 in C).

Figure 4 shows an efficient implementation of the string-matching algorithm. Note that the $w - m$ leftmost bits of the unsigned variables are always 1. This is to avoid one more operation to set them to 0 in the main loop. Another implementation is possible using *op* as a bitwise logical *and* operation, and complementing the value of $T[x]$ for all $x \in \Sigma$.

Figure 4. Shift-Or algorithm for string matching

```
#define WORD     32      /* Word size in bits                */
#define EOS       0      /* End of string                    */
#define B         1      /* Number of bits per state         */
#define MAXSYM 128       /* Size of the alphabet (ASCII)     */

Search( text, pattern ) /* Shift-Or Algorithm for String Searching */
register char *text;
char *pattern;
{
    register unsigned int state, lim, first, initial;
    unsigned int T[MAXSYM];
    int i, j, matches;

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

Experimental results for searching 100 times for all possible matches of a pattern in a 50,000-character English text (a legal document) are presented in Table 2. For each pattern, a prefix from length 2 to 10 was used (for example, for "representative" the queries were {"re", "rep", . . . , "representa"}. The patterns were chosen so that each first letter had a different frequency in English text (from most to least frequent). The timings are in seconds and they have an absolute error bounded by 0.5 seconds. They include the preprocessing time in all cases.

The algorithms implemented are Boyer-Moore, as suggested by Horspool [19] (BMH), which according to Baeza-Yates [3], is the fastest practical version of this algorithm; Knuth-Morris-Pratt, as suggested by their authors [24], and our new algorithm (SO) using the KMP idea (first scan for the first character of the pattern.) This version is shown in Figure 4. Note that SO and KMP are dependent on the frequency of the first letter of the pattern in the text, and that BMH depends on the pattern length. Another possible implementation is to combine the SO algorithm with the BMH algorithm, searching from left to right, but jumping as in the BMH algorithm. This idea is used in a KMP-BMH hybrid algorithm [7].

From Table 5 we can see that SO outperforms KMP, being between 40% and 50% faster. This is true only for $m \leq w$. Also it is faster than BMH for patterns of length smaller than 4 to 9, depending on the pattern (Horspool [19] mentions that BMH should be used for $m > 5$). Compared with the *grep* program (Berkeley Unix operating system) our algorithm is between 45% and 30% faster (see Table 2), in spite of

grep using faster input routines (low level).

Figure 5 shows the execution time while searching 1,000 words chosen at random from the same English text. In this figure, data for the brute-force algorithm (try every possible position) is also included. The SO algorithm is faster than the Knuth-Morris-Pratt algorithm for $m < w$, and is faster than the Boyer-Moore algorithm for small m. Note that the SO algorithm requires the **same** time for patterns with classes.

Figures 6 to 8 show the implementation for pattern matching with at most k mismatches and word size 32 bits, using $O(\log k)$ bits per state. Since the code is similar to the exact string-matching case, this algorithm is slightly slower and is independent of k for $m = O(w/\log k)$. Table 6 shows the simulation results for searching 1,000 strings of the same English text as before for two values of k and small values of m. We include data for three algorithms: a naive algorithm that tries every possible position; the Landau-Vishkin algorithm (using a window of size $O(m^2)$ to process the text, instead of the $O(mn)$ table suggested in the original paper [25]) and the shift-add algorithm for a word size of 32 bits (using 4 bits if $m < 9$, or $O(\log k)$ bits otherwise). Data for Galil and Giancarlo's algorithm are not included because it was even slower than the Landau-Vishkin algorithm. According to these results, the shift-add algorithm is clearly faster, even if multiple word arithmetic is necessary for larger values of m.

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

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

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

Table 2. Maximum pattern length (m) for a 32-bit word depending on k

Bits per state	$k, m - k$		m
	$b = \lceil \log_2 k \rceil + 1$	$b = \lceil \log_2(k+1) \rceil$	
1	0	0	32
2	1	1–2	16
3	2–3	3–5	10
4	4		8

Table 3. Example for string searching with mismatches

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

Table 4. Improved string searching with mismatches

$x \leftarrow ((state \;\&\; 11110_{(m-1)5+1}...10_6 11110) >> 1) \;\&\; state$
$x \leftarrow x \;\&\; (x >> 2)$
$x \leftarrow x \;\&\; (x >> 1)$
$state \leftarrow state + (T[char] \;\&\; x)$

Table 5. Experimental results for prefixes of 4 different patterns (time in seconds)

m	Pattern: epresentative				Pattern: representative			
	BMH	KMP	SO	grep	BMH	KMP	SO	grep
2	36.5	24.4	15.8	21.3	23.6	15.5	13.2	17.9
3	25.2	24.3	15.7	21.1	16.2	15.0	13.0	19.8
4	20.5	24.5	15.6	21.6	12.6	15.0	13.1	20.0
5	17.3	24.3	15.8	21.5	11.0	15.2	13.1	19.6
6	15.3	24.4	15.9	22.0	9.6	15.1	13.4	19.3
7	13.2	24.3	15.7	21.5	9.0	15.3	13.1	19.6
8	12.5	24.4	15.6	21.9	7.9	15.3	13.3	19.3
9	11.6	24.4	15.8	22.0	7.5	15.3	13.3	19.4
10	11.2	24.3	15.8	21.8	7.1	15.4	13.0	19.8
m	Pattern: legislative				Pattern: kinematics			
	BMH	KMP	SO	grep	BMH	KMP	SO	grep
2	37.7	21.0	11.9	19.1	35.2	19.0	10.4	19.1
3	25.6	21.0	12.3	19.3	24.9	19.0	10.5	18.1
4	19.9	20.9	11.8	20.0	19.9	18.8	10.4	18.4
5	16.5	20.6	11.7	19.6	16.7	19.0	10.4	18.5
6	14.3	20.6	11.6	19.2	14.3	19.1	10.4	18.6
7	12.9	20.5	11.8	19.8	13.0	19.0	10.4	18.7
8	12.0	20.6	12.0	19.7	12.2	19.0	10.4	17.9
9	11.2	20.7	12.1	19.7	10.8	19.0	10.6	18.2
10	10.3	20.9	11.8	19.7	10.0	19.1	10.5	18.8

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

Final Remarks

We have presented a simple class of algorithms that can be used for string matching and some other kinds of patterns, with or without mismatches. These are the first practical algorithms for string matching with classes and/or mismatches.

The time complexity achieved is linear for pattern lengths smaller than the word size in bits, and this is the case in most applications. For longer patterns, we need to implement integer arithmetic of the precision needed using more than a word per number. Still, if the number of words per number is small, our algorithm is a good practical choice for string matching with classes and/or mismatches. If we have large m, we could also use this algorithm to find a partial match, and then verify that it is a true match with a simpler algorithm. The running time will be linear if we have at most $O(n/m)$ matches.

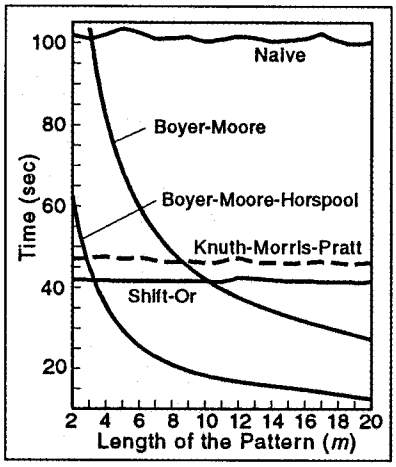
Using VLSI technology to have a chip that uses a register of 64 or 128 bits that implements this algorithm for a stream of text, faster searching times can be achieved.

The applications of these algorithms are restricted to main memory (text editing for example), or to text databases where a very coarse granularity index is provided and pattern matching is done with the granules or within partial answers.

This type of algorithm can also be used for other matching problems, for example for patterns of the form (*set of patterns*)Σ^*(*set of patterns*) (see [26]) where each set is one or more strings with or without classes, and Σ^* represents 0 or more arbitrary characters.

Future research is to extend this to other kinds of errors, for example the transposition of two characters (this operation also maintains the length of the string); and the design of a hardware implementation.

Figure 5. Experimental results for searching 1,000 strings in English text



Historical Note added in Proof

This work was done while both au-

Table 6. Experimental results (in seconds) for searching 1,000 patterns with at most k mismatches for some pattern lengths (m)

	$k=1$			$k=2$			$k=3$		
m	Naive	LV	SA	Naive	LV	SA	Naive	LV	SA
2	576.4	5618.5	214.6						
3	630.1	5660.4	207.5	774.6	6989.9	207.1			
4	628.0	5663.8	205.4	832.6	7040.1	206.1	982.5	8616.9	207.6
5	629.8	5668.6	205.6	842.6	7047.7	205.9	1043.6	8676.6	205.9
6	626.5	5660.3	205.6	843.7	7046.2	205.9	1055.4	8684.6	205.9
7	628.1	5669.0	205.9	845.0	7053.9	205.4	1059.1	8691.9	205.7
8	636.9	5665.5	206.6	845.5	7055.7	207.7	1060.4	8689.5	212.1

Figure 6. String matching with at most k mismatches

```
unsigned int mask, lim, ovmask; /* Preprocessing variables   */
unsigned int T[MAXSYM];
int B;                  /* number of bits per state          */
int type;               /* kind of search (MATCH or MISMATCH)*/

Patmat( k, pattern, text ) /* Pattern matching with k mismatches */
int k;                     /* (WORD=32, MAXSYM=128, EOS=0) */
char *pattern, *text;
{
   int m;

   m = strlen(pattern); type = MISMATCH; /* count mismatches */
   if( 2*k > m ) /* Pattern matching with at least m-k matches */
   {
      type = MATCH; k = m-k; /*count matches */
   }
   B = clog2(k+1) + 1; /* ceiling of log base 2 of k+1 */
   if( m > WORD/B ) Error( "Search does not work for this case" );
   /* Preprocessing */
   Preprocessing( pattern, m, k );
   /* Search */
   return( Search( text ));
}

int clog2( x ) int x; { /* Ceiling of log2(x)          */
int i = 0;       /* log2(x) for x<=0 returns 0 */

while( x > (1 << i)) i++;
return(i);
}
```

thors were at the University of Waterloo, and it is part of the Ph.D. dissertation of the first author [6, 10]. This article was first submitted in 1989, and revised in early 1990. Because of the publication delay several new references are missing, and the experimental results are partially outdated. In particular, several practical improvements to Boyer-Moore type algorithms have been published [14, 20, 27, 28]. During this time, Wu and Manber extended very nicely the basic algorithm presented in this article to include approximate string searching [29, 30]. Their article appears in this issue of *Communications of the ACM*. Recently, another technique for string searching which is not based on comparisons has been presented [11]. These results show that the full potential of practical noncomparison-based text-searching algorithms is yet to be explored. ◻

References

1. Abrahamson, K. Generalized string matching. *SIAM J Comput. 16* (1987), 1039–1051.
2. Aho, A.V. and Corasick, M. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18,* 6 (June 1975), 333–340.
3. Baeza-Yates, R. Improved string searching. *Software-Pract. and Exper. 19,* 3 (1989), 257–271.
4. Baeza-Yates, R. String searching algorithms. In *Information Retrieval: Algorithms and Data Structures,* Chapt. 10, W. Frakes and R. Baeza-Yates, Eds., pp. 219–240. Prentice Hall, Englewood Cliffs, N.J., 1992.
5. Baeza-Yates, R. Text retrieval: Theory and practice. In *Twelfth IFIP World Computer Congress* (Madrid, Spain, Sept. 1992).
6. Baeza-Yates, R.A. Efficient text searching. Ph.D. thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Res. Rep. CS-89-17.
7. Baeza-Yates, R.A. String searching algorithms revisted. In *Workshop in Algorithms and Data Structures,* F. Dehne, J.-R. Sack, and N. Santoro, Eds., pp. 75–96 (Ottawa, Canada, Aug. 1989). Springer Verlag Lecture Notes on Computer Science 382.

Figure 7. Preprocessing for string matching with mismatches

```
Preprocessing( pat, m, k )
char *pat;
int m, k;
{
   int i;

   lim = k << ((m-1)*B);
   ovmask = 0;
   for( i=1; i<=m; i++ ) ovmask = (ovmask << B) | (1 << (B-1));
   if( type == MATCH )
      for( i=0; i<MAXSYM; i++ ) T[i] = 0;
   else
   {
      lim += 1 << ((m-1)*B);
      for( i=0; i<MAXSYM; i++ ) T[i] = ovmask >> (B-1);
   }
   for( i=1; *pat != EOS; i <<= B )
   {
      if( type == MATCH )
         T[*pat] += i;
      else
         T[*pat] &= ~i;
      pat++;
   }
   if( m*B == WORD ) mask = ~0;
   else              mask = i - 1;
}
```

Figure 8. Search phase of string matching with *k* mismatches

```
int Search( text ) /* Search phase */
char *text;
{
   unsigned int state, overflow;
   int matches;
   matches = 0;
   /* Initial state */
   if( type == MATCH ) { state = 0; overflow = 0; }
   else { state = mask & ~ovmask; overflow = ovmask; }
   for( ; *text != EOS; text++ )
   {
      state = ((state << B) + T[*text]) & mask;
      overflow = ((overflow << B) | (state & ovmask)) & mask;
      state &= ~ovmask;
      if( type == MATCH )
      {
         if( (state | overflow) >= lim )
            matches++; /* Match with more than m-k errors */
      }
      else if( (state | overflow) < lim )
         matches++; /* Match with (state>>(m-1)*B) errors */
   }
   return( matches );
}
```

8. Baeza-Yates, R. and Gonnet, G.H. Fast string matching with mismatches. *Inf. Comput.* (1992). To be published. Also as Tech. Rep. CS-88-36, Dept. of Computer Science, University of Waterloo, 1988.
9. Baeza-Yates, R. and Gonnet, G.H. A new approach to text searching. In *Proceedings of Twelfth ACM SIGIR* (Cambridge, Mass., June 1989) pp. 168–175. (Addendum in ACM SIGIR Forum, W. 23, Numbers 3, 4, 1989, p. 7.).
10. Baeza-Yates, R. and Gonnet, G.H. New algorithm for pattern matching with and without mismatches. Tech. Rep. CS-88-37, Department of Computer Science, University of Waterloo, Ontario, Canada, 1988.
11. Baeza-Yates, R.A. and Perleberg, C.H. Fast and practical approximate pattern matching. In *Proceedings of Third Conference on Combinatorial Pattern Matching* (Tucson, Ariz., Apr. 1992) pp. 182–189.
12. Boyer, R. and Moore, S. A fast string searching algorithm. *Commun. ACM 20* (1977), 762–772.
13. Commentz-Walter, B. A string matching algorithm fast on the average. In *ICALP, Lecture Notes in Computer Science,* vol. 6. Springer-Verlag, 1979, pp. 118–132.
14. *Commun. ACM 35,* 4 (Apr. 1992). Technical correspondence. Notes on a very fast substring search algorithm. 132–137.
15. Fischer, M. and Paterson, M. String matching and other products. In *Complexity of Computation,* R. Karp, Ed., (*SIAM-AMS Proceedings 7*), pp. 113–125. American Mathematical Society, Providence, R.I., 1974.
16. Galil, Z. and Giancarlo, R. Improved string matching with k mismatches. *SIGACT News 17* (1986), 52–54.
17. Gonnet, G.H. Unstructured data bases or very efficient text searching. In *ACM PODS 2,* Atlanta, Ga., Mar. 1983, pp. 117–124.
18. Gonnet, G.H. and Baeza-Yates, R. *Handbook of Algorithms and Data Structures—In Pascal and C.* Second ed. Addison-Wesley, Wokingham, UK, 1991.
19. Horspool, R.N. Practical fast searching in strings. *Softw.—Pract. Exper. 10* (1980), 501–506.
20. Hume, A. and Sunday, D.M. Fast string searching. *Softw.—Pract. Exper. 21,* 11 (Nov. 1991), 1221–1248.
21. Karp, R. and Rabin, M. Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop. 31* (1987), 249–260.
22. Kernighan, B. and Ritchie, D. *The C Programming Language.* Prentice Hall, Englewood Cliffs, N.J., 1978.
23. Knuth, D.E. *The Art of Computer Programming: Sorting and Searching,* vol. 3. Addison-Wesley, Reading, Mass., 1973.
24. Knuth, D.E., Morris, J. and Pratt, V. Fast pattern matching in strings. *SIAM J Comput. 6* (1977), 323–350.
25. Landau, G. and Vishkin, U. Efficient string matching with k mismatches. *Theoretical Comput. Sci. 43* (1986), 239–249.
26. Pinter, R. Efficient string matching with don't-care patterns. In *Combinatorial Algorithms on Words,* A. Apostolico and Z. Galil, Eds., vol. F12 of *NATO ASI Series,* Springer-Verlag, 1985, pp. 11–29.
27. Smith, P.D. Experiments with a very fast substring search algorithm. *Softw.—Pract. Exper. 21,* 10 (Oct. 1991), 1065–1074.
28. Sunday, D.M. A very fast substring search algorithm. *Commun. ACM 33,* 8 (Aug. 1990), 132–142.
29. Wu, S. and Manber, U. Fast text searching with errors. Tech. Rep. TR-91-11, Department of Computer Science, University of Arizona, Tucson, Ariz., June 1991.
30. Wu, S. and Manber, U. Agrep—a fast approximate pattern-matching tool. In *Proceedings of USENIX Technical Conference* (Jan. 1992, San Francisco, Calif.), pp. 153–162.

CR Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*retrieval models, search process, selection process*

General Terms: Algorithms

Additional Key Words and Phrases: String matching, text searching

About the Authors:

RICARDO BAEZA-YATES is an associate professor at the Universidad de Chile in Santiago, Chile. Current research interests include algorithms and data structures, text retrieval, graphical user interfaces, and object-oriented programming. **Author's Present Address:** Universidad de Chile, Blanco Encalada 2120, Depto. de Ciencias de la Computacion, Santiago, Chile; email: rbaeza@dcc.uchile.cl

GASTON D. GONNET is a professor at the Swiss Technological Institute in Zurich, Switzerland. Current research interests include algorithms and data structures, algebraic manipulation, text retrieval, computational biology, and programming languages design. **Author's Present Address:** Informatik, Swiss Technological Institute in Zurich, Switzerland; email: gonnet@inf.ethz.ch