A New Approach to Text Searching

(Preliminary version)

Ricardo A. Baeza-Yates
Gaston H. Gonnet

Centre for the New O.E.D.
& Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1 *

Abstract

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

1 Introduction

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

The main idea is to represent the state of the search as a number, and each search step costs a small number of arithmetic/logical operations, provided that the numbers are big enough to represent all possible states of the search. Hence, for small patterns, we have a $O(n)$ time algorithm using $O(|\Sigma|)$ extra space and $O(m+|\Sigma|)$ preprocessing time, where Σ denotes the alphabet.

*The work of the first author was supported by a scholarship from the Institute for Computer Research of the University of Waterloo and by the University of Chile and that of the second author by a Natural Sciences and Engineering Research Council of Canada Grant No. A-3353.

For string matching, empirical results show that the new algorithm compares favourably with the Knuth-Morris-Pratt (KMP) algorithm [11] for any pattern length and the Boyer-Moore (BM) algorithm [4] 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, generalizing this to any finite class of symbols or their complement.

The main properties of this class of algorithms are:

- Simplicity: the preprocessing and the search are very simple, and only bitwise logical operations, shifts and additions are used.
- Real time: the time-delay to process one text character is bounded by a constant.
- No buffering: neither the text nor the pattern need to be stored.

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

All these properties indicates that this class of algorithms is suitable for hardware implementation; hence we believe that this new approach is a valuable contribution to all applications dealing with text searching. The motivation behind our work is the work done for New Oxford English Dictionary project at the University of Waterloo.

2 A Numerical approach to String Matching

After the discovery of linear time string matching algorithms [11,4] a lot of research was done in the area. Our algorithm is based on finite automata theory [11] and also exploits the fact that in practical applications the alphabet is finite [4].

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

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

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

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

For string matching we need only 1 bit (that is $b = 1$), where s_i is 0 if the last i characters have matched or 1 if not. We have to report a match if s_m is 0, or equivalently if $state < ...1110^{m-1}$.

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

- shift the vector state b bits to the left to reflect that we have advanced in the text one position. In practice, this sets the initial state of s_1 to be 0 by default.

- update the individual states according with the new character. For this, we use a table T that is defined by preprocessing the pattern with one entry per alphabet symbol, and an operator *op* that, given the old vector state and the table value, gives the new state. Note that this works only if the effect of the operator in the individual state s_i does not produce a carry that will affect state s_{i+1}.

Then, each search step is:

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

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

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

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

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

Example : Let $\{a, b, c, d\}$ be the alphabet, and *ababc* the pattern. Then, if $b = 1$, the entries for the table T are:

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

The choice for *op* in the case of string matching is almost unique: a bitwise logical or. We finish the example, by searching the first occurrence of *ababc* in the text *abdababab c*.

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

text :         a     b     a     b     c
T[x] :         11010 10101 11010 10101 01111
state: 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 2 and 4. The match at the end of the text is indicated by the value 0 in the leftmost bit of the state of the search. □

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

For each kind of patterns or searching problem, we could adequately choose b and *op*. A similar idea was presented by Gonnet [8] applied to searching the signatures of a text.

3 String Matching with Classes

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

- x: a character from the alphabet.

- Σ: a don't care symbol (matches any symbol).

- [*characters*]: a class of characters, where we allow ranges (for example *a..z*).

- $\neg C$: the complement of a character or class of characters C. That is, matches any character that not belongs to this class.

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

String matching with don't care patterns was addressed before in Fischer and Paterson [6] achieving

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

asymptotic search time, and also in Pinter [13] 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 also a lot easier to implement.

Attempts to adapt the KMP algorithm to this case have failed [6,13], and for the same reason the BM algorithm as presented in Knuth *et al* [11] cannot solve this problem. It is possible to use the Horspool version of the BM algorithm [9], 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 [10] 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 slower in practice (because of the use of multiplications).

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

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

To maintain $O(\lceil \frac{mb}{w} \rceil (m + |\Sigma|))$ preprocessing time (instead of $O(\lceil \frac{mb}{w} \rceil m|\Sigma|)$ time), where m is now the size of the description of the pattern (and not its length), we use the complement of the class for don't care symbols and complements. The search time remains the same.

4 Pattern Matching with Mismatches

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

Landau and Vishkin [12] 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 [7] improve this algorithm to $O(kn + m \log m)$ time and $O(m)$ space. This algorithm is practical for small k. However, if $k = O(m)$, it is not so. Other approaches to this problem are presented in [3].

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

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

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

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

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

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

bits. At each step we record the overflow bits in an overflow state, and we reset the overflow bits of all individual states (in fact, we only have to do this each k steps, but it is not practical to get in all that trouble). Note that if $k > m/2$, then we count matches. The only problem for this case, is that is not possible to tell how many errors there are in a match. Table 1 shows up to what m we can use for a 32 bits word.

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

Table 1: Maximum pattern length (m) for a 32 bits word depending on k.

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

5 Multiple Patterns

We consider in this section the problem of more than one pattern, for patterns with classes (also we can extend this to mismatches). To denote the union symbol we use "|", for example $p_1|p_2$ searches for the pattern p_1 or the pattern p_2.

The KMP algorithm and the BM algorithm had been extended already to this case (see [1] and [5] respectively), achieving a worst case time of $O(n+m)$, where m is the total length of the set of patterns.

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

6 Implementation

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

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

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

Figure 1 shows an efficient implementation of the string matching algorithm. Another implementation is possible using *op* as a bitwise logical *and* operation, and complementing the value of T_x for all $x \in \Sigma$.

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

The algorithms implemented are Boyer-Moore, as suggested by Horspool [9] (or BMH), which, according to Baeza-Yates [2], is the fastest practical version of this algorithm; Knuth-Morris-Pratt, as suggested by their authors [11] (or KMP_1, and as given by Sedgewick [14] (or KMP_2); and our new algorithm as presented in Figure 1 (SO_1), and another version using the KMP_1 idea (SO_2) (that is, do not use the algorithm until we see the first character of the pattern). The changes needed for the later case (using structured programming!) are shown in Figure 2. Note that SO_1 and KMP_2 will be independent of the pattern length, that SO_2 and KMP_1 will be dependent of the frequency of the first letter of the pattern in the text, and that BMH depends on the pattern length.

From Table 2 we can see that SO_2 outperforms KMP_1, being between a 40% and 50% faster. Also it is faster than BMH for patterns of length smaller than 4 to 9, depending on the pattern.

Figure 3 shows the preprocessing phase for patterns with classes, using "^" as the complement character and "\" as the escape character. The search phase remains as before. The search time for this class of patterns is the same as the search time for a string of the same length.

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

Figure 5 shows the changes needed for the case where we use $O(\log k)$ bits per state.

For multiple patterns, the preprocessing is very similar to the one in Figure 3. The only change in the search phase is the match testing condition:

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

where mask has a bit with value 1 in the adequate position for each pattern. Note that this indicates that a pattern *ends* at the current position, and it is not possible to say where the pattern starts without wasting $O(\lceil\frac{mb}{w}\rceil sM)$ time, being M the number of matches and s the number of patterns.

7 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. The time complexity achieved is linear for small patterns, and this is the case in most applications. For longer patterns, we need to implement integer arithmetic of the precision needed using more than a word per number. Still, if the number of words per number is small, our algorithm is a good practical choice. Using VLSI technology to have a chip that uses a register of 64 or 128 bits that implements this algorithm for a stream of text, faster search time can be achieved.

The applications of these algorithms are restricted to main memory, or to text data bases where a very coarse granularity index is provided and pattern matching is done within the granules.

This type of algorithms can also be used for other matching problems, for example mismatches with different costs (see [3]) or for patterns of the form $(set\ of\ patterns)\Sigma^*(set\ of\ patterns)$ (see [13]).

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

Figure 1: Shift-Or algorithm for string matching.

	Pattern: epresentative					Pattern: representative				
m	BMH	KMP_1	KMP_2	SO_1	SO_2	BMH	KMP_1	KMP_2	SO_1	SO_2
2	36.5	24.4	58.7	30.2	15.8	23.6	15.5	49.9	30.2	13.2
3	25.2	24.3	59.0	30.2	15.7	16.2	15.0	50.2	30.4	13.0
4	20.5	24.5	58.7	30.2	15.6	12.6	15.0	50.1	30.2	13.1
5	17.3	24.3	58.8	30.4	15.8	11.0	15.2	50.1	30.4	13.1
6	15.3	24.4	58.7	30.3	15.9	9.6	15.1	50.9	30.6	13.4
7	13.2	24.3	58.6	30.1	15.7	9.0	15.3	50.8	30.5	13.1
8	12.5	24.4	58.6	30.4	15.6	7.9	15.3	50.7	30.6	13.3
9	11.6	24.4	59.7	30.1	15.8	7.5	15.3	50.5	30.7	13.3
10	11.2	24.3	58.3	30.1	15.8	7.1	15.4	50.1	30.2	13.0

	Pattern: legislative					Pattern: kinematics				
m	BMH	KMP_1	KMP_2	SO_1	SO_2	BMH	KMP_1	KMP_2	SO_1	SO_2
2	37.7	21.0	58.2	30.6	11.9	35.2	19.0	57.6	30.2	10.4
3	25.6	21.0	58.6	31.1	12.3	24.9	19.0	57.4	30.1	10.5
4	19.9	20.9	57.8	30.4	11.8	19.9	18.8	57.4	29.9	10.4
5	16.5	20.6	57.8	30.1	11.7	16.7	19.0	57.4	30.0	10.4
6	14.3	20.6	58.0	30.2	11.6	14.3	19.1	57.6	30.1	10.4
7	12.9	20.5	57.5	30.1	11.8	13.0	19.0	57.5	30.1	10.4
8	12.0	20.6	57.9	30.3	12.0	12.2	19.0	57.6	30.0	10.4
9	11.2	20.7	57.7	30.3	12.1	10.8	19.0	57.3	30.1	10.6
10	10.3	20.9	58.2	30.3	11.8	10.0	19.1	57.5	30.2	10.5

Table 2: Experimental results for prefixes of 4 different patterns (time in seconds)

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

Figure 2: Shift-Or algorithm for string matching (trickier version).

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

Figure 3: Preprocessing for Patterns with Classes.

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

Figure 4: Pattern Matching with at most k mismatches (simpler version).

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

Figure 5: String Matching with at most k mismatches.