CHAPTER 3

STRING-MATCHING ALGORITHMS
3.0 OUTLINE OF THIS CHAPTER

This chapter is concerned with string matching methods for locating patterns occurring as a sub-string of a particular string. Such keywords searches are a common requirement in, for example, word processing and information retrieval applications. This chapter discusses the most popular string matching algorithms.

- Brute-Force Algorithm
- Karp-Rabin Algorithm
- Knuth-Morris-Pratt Algorithm
- Boyer-Moore Algorithm
- Quick Search Algorithm

3.1 INTRODUCTION

The general approach for looking for a pattern in a file that is stored in its compressed form is first decompressing and then applying one of the known pattern matching algorithms in the decoded file. In many cases, however, in particular on the Internet, files are stored in their original form, for if they were compressed, the host computer would have to provide memory space for each user in order to store the decoded file. This requirement is not reasonable, as many user scan simultaneously quest the same information reservoir which will demand an astronomical quantity of free memory. Another possibility is transferring the compressed files to the personal computer of the user, and then decoding the files. However, we then assume that the user knows the exact location of the information he is looking for; if this is not the case, much unneeded information will be transferred.

There is therefore a need to develop methods for directly searching within a compressed file. For a given text $S$ and pattern $P$ and complementary encoding and decoding functions $E$ and $D$, our aim is to search $E(P)$ in $E(S)$, rather than the usual approach which searches for the pattern $P$ in the decompressed text $D(E(S))$. But this is not always
straightforward, since an instance of $E(P)$ in the compressed text is not necessarily the encoding of instance of $P$ in the original text $S$. This so-called compressed matching problem has been introduced by Amir and Benson [47]. The algorithms proposed in chapter 4 are useful for searching $E(P)$ in $E(S)$, with any conventional string-matching algorithm discussed in this chapter.

3.2 STRING-MATCHING ALGORITHMS

String matching consists of finding one, or more generally, all the occurrences of a pattern in a text. The pattern and the text are both strings built over a finite alphabet (a finite set of symbols). Each algorithm describes here outputs all occurrences of the pattern in the text. The pattern is denoted by $P = P[0...m-1]$; its length is equal to $m$. The text is denoted by $S = S[0...n-1]$; its length is equal to $n$. The alphabet is denoted by $\Sigma$ and its size equal to $\sigma$.

String-matching algorithms work as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern — this specific work is called an attempt or a scan — and after a whole match of the pattern or after a mismatch they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. This is called the scan and shift mechanism. Each attempt is associated with the position $i$ in the text when the pattern is aligned with $S[i...i+m-1]$.

The brute force algorithm consists of checking, at all positions in the text between 0 and $n-m$, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern exactly one position to the right. This is the simplest algorithm, which is described in Figure 3.1.

The time complexity of the brute force algorithm is $O(mn)$ in the worst case but its behavior in practice is often linear on specific data.

Four categories arise: the most natural way to perform the comparisons is from left to right, which is the reading direction; performing the comparisons from right to left generally leads to the best algorithms in practice; the best theoretical bounds are reached when comparisons are done in a specific order; finally there exist some algorithms for
which the order in which the comparisons are done is not relevant (such is the brute force algorithm)

```c
void BF(char *s, char *p, int n, int m)
{
    int i, j;
    /* Searching */
    for (i=0; i <= n-m; i++)
    {
        j=0;
        while (j < m && s[i+j] == p[j])
            j++;
        if (j >= m)
            OUTPUT(i);
    }
}
```

Figure 3.1 The Brute Force string-matching algorithm.

### 3.2.1 From left to right

Hashing provides a simple method that avoids the quadratic number of character comparisons in most practical situations and that runs in linear time under reasonable probabilistic assumptions. It has been introduced by Harrison and later fully analyzed by Karp and Rabin[66]. Assuming that the pattern length is no longer than the memory-word size of the machine, the Shift-Or algorithm is an efficient algorithm to solve the exact string-matching problem and it adapts easily to a wide range of approximate string-matching problems.

The first linear-time string matching algorithm is from Morris and Pratt [67]. It has been improved by Knuth, Morris, and Pratt [7]. The search behaves like a recognition process by automaton and a character of the text is compared to a character of the pattern no more than \( \log_\phi (m+1) \) (\( \phi \) is the golden ratio \((1+\sqrt{5})/2\)). Hancart proved that this delay of a related algorithm discovered by Simon makes no more than \( 1 + \log_2 m \) comparisons per text character. Those three algorithms perform at most \( 2n-1 \) text character comparisons in the worst case.

The search with a Deterministic Finite Automaton performs exactly \( n \) text character inspections but it requires an extra space in \( O(m \times \sigma) \). The Forward Dawg Matching algorithm performs exactly the same number of text character inspections using the suffix automaton of the pattern.
The Apostolico-Crochemore algorithm is a simple algorithm which performs $3/2n$ text character comparisons in the worst case.

The Not So Naive algorithm is a very simple algorithm with a quadratic worst case time complexity but it requires a preprocessing phase in constant time and space and is slightly sub-linear in the average case.

### 3.2.2 From right to left

The Boyer-Moore algorithm[6] is considered as the most efficient string matching algorithm in usual applications. A simplified version of it (or the entire algorithm) is often implemented in text editors for the “search” and “substitute” commands. Cole proved that the maximum number of character comparisons is tightly bounded by $3n$ after the preprocessing for non-periodic patterns. It has a quadratic worst case time for periodic patterns.

Several variants of the Boyer-Moore algorithm avoid its quadratic behavior. The most efficient solutions in term of number of character comparisons have been designed by Apostolico and Giancarlo, Crochemore et alii (TurboBM) and Colussi (Reverse Colussi). Empirical results show that variations of the Boyer-Moore algorithm and algorithms based on the suffix automaton by Crochemore et alii (Reverse Factor and Turbo Reverse Factor) or the suffix oracle by Crochemore et alii (Backward Oracle Matching) are the most efficient in practice.

The Zhu-Takaoka and Berry-Ravindran algorithms are variants of the Boyer-Moore algorithm which require an extra space in $O(\sigma^2)$

### 3.2.3 In a specific order

The two first linear optimal space string-matching algorithms are due to Galil-Seiferas and Crochemore-Perrin (Two Way). They partition the pattern in two part, they first search for the right part of the pattern from left to right and then if no mismatch occurs they search for the left part.

The algorithms of Colussi and Galil-Giancarlo partition the set of pattern positions into two subsets. They first search for the pattern characters which positions are in the first subset from left to right and then if no mismatch occurs they search for the remaining
characters from left to right. The Colussi algorithm is an improvement over the Knuth-Morris-Pratt algorithm and performs at most \( \frac{3}{2}n \) text character comparisons in the worst case. The Galil-Giancarlo algorithm improves the Colussi algorithm in one special case which enables it to perform at most \( \frac{4}{3}n \) text character comparisons in the worst case.

Sunday’s Optimal Mismatch and Maximal Shift algorithms sort the pattern positions according their character frequency and their leading shift respectively.

Skip Search, KmPSkip Search and Alpha Skip Search algorithms by Charras et alii use buckets to determine starting positions on the pattern in the text.

### 3.2.4 In any order

The Horspool algorithm is a variant of the Boyer-Moore algorithm. It uses only one of its shift functions and the order in which the text character comparisons are performed is irrelevant. This is also true for other variants such as the Quick Search algorithm of Sunday[68], Tuned Boyer Moore of Hume and Sunday, the Smith algorithm and the Raita algorithm.

### 3.3 KARP-RABIN ALGORITHM

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position of the text whether the pattern occurs, it seems to be more efficient to check only if the portion of the text aligned with the pattern “looks like” the pattern. In order to check the resemblance between these portions a hashing function is used. To be helpful for the string-matching problem the hashing function should have the following properties:

- efficiently computable,
- highly discriminating for strings,
- \( \text{hash} (s[i+1..i+m]) \) must easily computable from \( \text{hash}(s[i+1..i+m-1]) \):
  \[
  \text{hash} (s[i+1..i+m]) = \text{rehash}(s[i],s[i+m],\text{hash}(s[i..i+m-1])).
  \]

For a word \( w \) of length \( k \), its symbols can be considered as digits, and we define \( \text{hash}(w) \) by:

\[
\text{hash}(w[0..k-1]) = (w[0] \times 2^{k-1} + w[1] \times 2^{k-2} + \ldots + w[k-1]) \mod q,
\]
where \( q \) is a large number.

Then, rehash has a simple expression

\[
\text{rehash}(a, b, h) = ((h - a \cdot d) \cdot 2 + b) \mod q,
\]

where \( d = 2^{k-1} \).

During the search for the pattern \( P \), it is enough to compare \( \text{hash}(p) \) with \( \text{hash}(s[i..i+m-1]) \) for \( 0 \leq i \leq n-m \). If an equality is found, it is still necessary to check the equality \( P = s[i..i+m-1] \) symbol by symbol.

In the algorithm of Figure 3.2 all the multiplications by 2 are implemented by shifts. Furthermore, the computation of the modulus function is avoided by using the implicit modular arithmetic given by the hardware that forgets carries in integer operations. So, \( q \) is chosen as the maximum value of an integer.

```c
#define REHASH(a, b, h) (((h-a*d)<<1)+b)
void KR(char *s, char *p, int n, int m)
{
    int hs, hp, d, i;
    /* Preprocessing */
    /* computes \( d = 2^{(m-1)} \) with the left-shift operator */
    d=1;
    for (i=1; i < m; i++)
        d<<=1;
    hs=hp=0;
    for (i=0; i < m; i++)
    {
        hp=((hp<<1)+p[i]);
        hs=((hs<<1)+s[i]);
    }
    /* Searching */
    for (i=m; i <= n; i++)
    {
        if (hs == hp && strcmp(s+i-m, p, m) == 0) OUTPUT(i-m);
        hs=REHASH(s[i-m], s[i], hs);
    }
}
```

**Figure 3.2** The Karp-Rabin string-matching algorithm.

The worst-case time complexity of the Karp-Rabin algorithm is quadratic in the worst case (as it is for the brute force algorithm) but its expected running time is \( O(m+n) \).

**Example 3.1:**

Let \( P = \text{ing} \).

Then \( \text{hash}(p) = 105 \cdot 22 + 110\cdot2 + 103 = 743 \) (symbols are assimilated with their ASCII codes).
3.4 KNUTH-MORRIS-PRATT ALGORITHM

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of the brute force algorithm, and especially on the way this latter algorithm wastes the information gathered during the scan of the text.

Let us look more closely at the brute force algorithm. It is possible to improve the length of shifts and simultaneously remember some portions of the text that match the pattern. This saves comparisons between characters of the text and of the pattern, and consequently increases the speed of the search.

Consider an attempt at position \( i \), that is, when the pattern \( P[0..m-1] \) is aligned with the window \( S[i..i+m-1] \) on the text. Assume that the first mismatch occurs between symbols \( S[i+j] \) and \( P[j] \) for \( 1 < j < m \). Then, \( S[i..i+j-1] = P[0..j-1] = u \) and \( a = S[i+j] \neq P[j] = b \). When shifting, it is reasonable to expect that a prefix \( v \) of the pattern matches some suffix of the portion \( u \) of the text. Moreover, to avoid another immediate mismatch, the letter following the prefix \( v \) in the pattern must be different from \( b \). The longest such prefix \( v \) is called the border \( u \) (it occurs at both ends of \( u \)). This introduces the notation: let \( \text{next}[j] \) be the length of the longest (proper) border of \( P[0..j-1] \) followed by a character \( c \) different from \( P[j] \). Then, after a shift, the comparisons can resume between characters \( S[i+j] \) and \( P[\text{next}[j]] \) without missing any occurrence \( P \) in \( S \), and avoiding a backtrack on the text (see Figure 3.3).

**Example 3.2:**
\[
S = \ldots \text{a b a b a a} \ldots \\
P = \begin{array}{ccccccc}
\text{a b a b a b a} \\
\text{a b a b a b a}
\end{array}
\]
Compared symbols are underlined. Note that the empty string is the suitable border of $ababa$. Other borders of $ababa$ are $aba$ and $a$.

The Knuth-Morris-Pratt algorithm is displayed in Figure 3.4. The table $next$ it uses is computed in $O(m)$ time before the search phase, applying the same searching algorithm to the pattern itself, as if $(S=P)$ (see Figure 3.5). The worst-case running time of the algorithm is $O(m+n)$ and it requires $O(m)$ extra space. These quantities are independent of the size of the underlying alphabet.

```c
void KMP(char *s, char *p, int n, int m)
{
    /* XSIZE is the maximum size of a pattern */
    int i, j, next[XSIZE];
    /* Preprocessing */
    PRE_KMP(p, m, next);
    /* Searching */
    i=j=0;
    while (i < n)
    {
        while (j > -1 && p[j] != s[i]) j=next[j];
        i++;
        if (j >= m)
        {
            OUTPUT(i-j);
            j=next[m];
        }
    }
}
```

Figure 3.4 The Knuth-Morris-Pratt string-matching algorithm.

3.5 BOYER-MOORE ALGORITHM

The Boyer-Moore algorithm is considered the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the “search” and “substitute” commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the bad-character shift and the good-suffix shift. They are based on the following observations.
```c
void PRE_KMP(char *p, int m, int next[]) {
    int i, j;
    i=0; j=next[0]=-1;
    while (i < m) {
        while (j > -1 && p[i] != p[j])
            j=next[j];
        i++; j++;
        if (i < m && p[i] == p[j])
            next[i]=next[j];
        else
            next[i]=j;
    }
}
```

Figure 3.5 Preprocessing phase of the Knuth-Morris-Pratt algorithm: computing \(next\).

Assume that a mismatch occurs between the character \(P[j] = b\) of the pattern and the character \(S[i+j] = a\) of the text during an attempt at position \(i\). Then, \(S[i+j+1\ldots i+m-1] = P[j+1\ldots m-1] = u\) and \(S[i+j] \neq P[j]\).

The good-suffix shift consists of aligning the segment \(S[i+j+1\ldots i+m-1] = P[j+1\ldots m-1]\) with its rightmost occurrence in \(P\) that is preceded by a character different from \(P[j]\) (see figure 3.6) if there exists no such segment, the shift consists of aligning the longest suffix \(v\) of \(S[i+j+1\ldots i+m-1]\) with a matching prefix of \(P\) (see figure 3.7).
Figure 3.8 Bad-character shift, \( a \) appears in \( P \).

Figure 3.9 Bad-character shift, \( a \) does not appears in \( P \).

Example 3.3:

\[
S = \ldots a b b a a b a a b a b a \ldots \\
P = a b b a a b a a b a b a \\
P = \quad a b b a a b b a b b a
\]

The shift is driven by the suffix \( abba \) of \( P \) found in the text. After the shift, the segment \( abba \) in the middle of \( S \) matches a segment of \( P \) as in figure 3.6. The same mismatch does not reoccur.

Example 3.4:

\[
S = \ldots a b b a a b a b b a a b b a \ldots \\
P = \quad b b a b b a b b a b a \\
P = \quad b b a b b a b b a b a
\]

The segment \( abba \) found in \( S \) partially matches a prefix of \( P \) after the shift, like in Figure 3.7.

The bad-character shift consists of aligning the text character \( S[i+j] \) with its rightmost occurrence in \( P[0 \ldots m-2] \) (see figure 3.8) If \( S[i+j] \) does not appear in the pattern \( P \), no occurrence of \( P \) in \( S \) can overlap the symbol \( S[i+j] \), then, the left end of the pattern is aligned with the character at position \( i+j+1 \) (see figure 3.9)
Example 3.5:
\[ S = \ldots a b c d \ldots \]
\[ P = c d a h g f e b c d \]
\[ P = \quad c d a h g f e b c d \]
The shift aligns the symbol \( a \) in \( P \) with the mismatch symbol \( a \) in the text \( S \) (Figure 3.8).

Example 3.6:
\[ S = \ldots a b c d \ldots \]
\[ P = c d h g f e b c d \]
\[ P = \quad c d h g f e b c d \]
The shift positions the left end of \( P \) right after the symbol \( a \) of \( S \) (Figure 3.9).

The Boyer-Moore algorithm is shown in Figure 3.10. For shifting the pattern, it applies the maximum between the bad-character shift and the good-suffix shift. More formally, the two shift functions are defined as follows. The bad-character shift is stored in a table \( bc \) of size \( \sigma \) and the good-suffix shift is stored in a table \( gs \) of size \( m+1 \). For \( a \in \sum \):

\[
bc[a] = \begin{cases} 
\min \{ j/1 \leq m \text{ and } x[m-1-j] = a \} & \text{if } a \text{ appears in } x \\
m & \text{otherwise}
\end{cases}
\]

```c
void BM(char *s, char *p, int n, int m)
{
    /* XSIZE is the maximum size of a pattern */
    /* ASIZE is the size of the alphabet */
    int i, j, gs[XSIZE], bc[ASIZE];
    /* Preprocessing */
    PRE_GS(p, m, gs);
    PRE_BC(p, m, bc);
    /* Searching */
    i=0;
    while (i <= n-m)
    {
        j=m-1;
        while (j >= 0 && p[j] == s[i+j])
            j--;
        if (j < 0)
            OUTPUT(i);
        i+=MAX(gs[j+1];
        bc[s[i+j]-m+j+1]; /* shift */
    }
}
```

Figure 3.10 The Boyer-Moore string-matching algorithm.

Let us define two conditions:

\[
\text{cond}_1(j, s): \text{ for each } k \text{ such that } j < k < m, s \geq k \text{ or } p[k-s] = p[k] \\
\text{cond}_2(j, s): \text{ if } s < j \text{ then } p[j-s] \neq p[j]
\]

Then, for \( 0 \leq j < m \):

\[
gs[j+1] = \min \{ s > 0 \text{ / cond}_1(j, s) \text{ and } \text{cond}_2(j, s) \text{ hold} \}
\]
and define $gs[0]$ as the length of the smallest period of $p$.

```c
void PRE_BC(char *p, int m, int bc[]) {
    /* ASIZE is the size of the alphabet */
    int j;
    for (j=0; j < ASIZE; j++) bc[j]=m;
    for (j=0; j < m-1; j++) bc[p[j]] = m-j-1;
}
```

Figure 3.11 Computation of the bad-character shift.

Tables $bc$ and $gs$ can be precomputed in time $O(m+\sigma)$ before the search phase and require an extra-space in $O(m+\sigma)$ (see Figures 3.12 and 3.11). The worst-case running time of the algorithm is quadratic. However, on large alphabets (relative to the length of the pattern) the algorithm is extremely fast. Slight modifications of the strategy yield linear-time algorithms.

When searching for $a^{m-1}b$ in $a^n$ the algorithm makes only $O(n/m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

```c
void PRE_GS(char *p, int m, int gs[]) {
    /* XSIZE is the maximum size of a pattern */
    int i, j, p, f[XSIZE];
    for (i=0; i <= m; i++) gs[i]=0;
    f[m]=j=m+1;
    for (i=m; i > 0; i--)
    {
        while (j <= m && p[i-1] != p[j-1])
        {
            if (!gs[j]) gs[j]=j-i;
            j=f[j];
        }
        f[i-1]=--j;
    }
    p=f[0];
    for (j=0; j <= m; j++)
    {
        if (!gs[j]) gs[j]=p;
        if (j == p) p=f[p];
    }
}
```

Figure 3.12 Computation of the good-suffix shift.
3.6 QUICK SEARCH ALGORITHM

The bad-character shift used in the Boyer-Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern, as it is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it only produces a very efficient algorithm in practice that is described now.

After an attempt where $P$ is aligned with $S[i..i+m-1]$, the length of the shift is at least equal to one. So, the character $S[i+m]$ is necessarily involved in the next attempt, and thus can be used for the bad-character shift of the current attempt. In the present algorithm, the bad-character shift is slightly modified to take into account the observation as follows ($a \in \Sigma$):

$$bc[a] = \begin{cases} \min\{j / 0 \leq m \text{ and } x[m-1-j] = a\} & \text{if } a \text{ appears in } x \\ m & \text{otherwise} \end{cases}$$

Indeed, the comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Figure 3.13 performs the comparisons from left to right. It is called Quick Search after its inventor and has a quadratic worst-case time complexity but a good practical behavior.

**Example 3.7:**

$S = \text{string - matching}$
$P = \text{ing}$
$P = \text{ing}$
$P = \text{ing}$
$P = \text{ing}$
$P = \text{ing}$

Quick Search algorithm makes 9 comparisons to find the two occurrences of \text{ing} inside the text of length 15.

For direct searching with simple text, the linear BF algorithm is a proper choice because it produces relatively good running time results despite its striking simplicity. In addition, the BF algorithm has no special memory requirements and needs no preprocessing or complex coding and thus can be surprisingly fast. But this algorithm shouldn’t use for the binary alphabet in applications such as image processing or software systems.
void QS(char *s, char *p, int n, int m)
{
    /* ASIZE is the size of the alphabet */
    int i, j, bc[ASIZE];
    /* Preprocessing */
    for (j=0; j < ASIZE; j++)
        bc[j]=m;
    for (j=0; j < m; j++)
        bc[p[j]]=m-j-1;
    /* Searching */
    i=0;
    while (i <= n-m)
    {
        j=0;
        while (j < m && p[j] == s[i+j])
            j++;
        if (j >= m)
            OUTPUT(i);
        i+=bc[s[i+m]]+1; /* shift */
    }
}

Figure 3.13 The Quick Search string-matching algorithm.

From the empirical evidence it can be concluded that the KR algorithm is linear in the number character comparisons but it has higher running time and it shouldn’t be used for pattern matching in strings. However, the main advantage of this algorithm lies in its extension to higher dimensional string matching. It may be used for pattern recognition and image processing and thus in the expanding field of computer graphics.

Despite its theoretical elegance, the KMP algorithm provides no significant speedup advantage over the BF algorithm in practice unless the pattern has highly repetitive subpatterns. However the KMP algorithm guarantees a linear bound and it is well suited to extensions for more difficult problems. It may be a good choice when the alphabet size is near the text size or when dealing with the binary alphabet.

Based on empirical results, it is clear that the QS algorithm is proved to be much faster algorithm in practice than the rest BM-like, suffix automata and bit-parallelism algorithms for large alphabets and short patterns. Therefore it is typically suited for search in the English alphabet. In addition, the BM algorithm is faster than its variations (such as BMH, QS, BMS and TBM) for small alphabets and long patterns. However, in theory BMS and QS are better algorithms than BM-like and suffix automata algorithms for short patterns and large alphabets [69].