(1984)
ivious
stores
(1936)
;ci. **53**
ıch. **17**
np. on
(1978)
ɔn any
ncepts
(1970)
·y and
c. 30th
ıs and
lations
st. Soc.

CHAPTER 5

Algorithms for Finding Patterns in Strings

Alfred V. AHO

AT & T Bell Laboratories, Murray Hill, NJ 07974, USA

Contents

HANDBOOK OF THEORETICAL COMPUTER SCIENCE
Edited by J. van Leeuwen
© Elsevier Science Publishers B.V., 1990

1. Introduction

String pattern matching is an important problem that occurs in many areas of science and information processing. In computing, it occurs naturally as part of data processing, text editing, term rewriting, lexical analysis, and information retrieval. Many text editors and programming languages have facilities for matching strings. In biology, string-matching problems arise in the analysis of nucleic acids and protein sequences, and in the investigation of molecular phylogeny. String matching is also one of the central and most widely studied problems in theoretical computer science.

The simplest form of the problem is to locate an occurrence of a keyword as a substring in a sequence of characters, which we will call the *input string*. For example, the input string *queueing* contains the "keyword" *ueuei* as a substring. Even for this problem, several innovative, theoretically interesting algorithms have been devised that run significantly faster than the obvious brute-force method. The problem becomes richer as we enlarge the class of patterns to include sets of keywords and regular expressions. This article examines the time–space trade-offs inherent in searching for occurrences of such patterns in text strings.

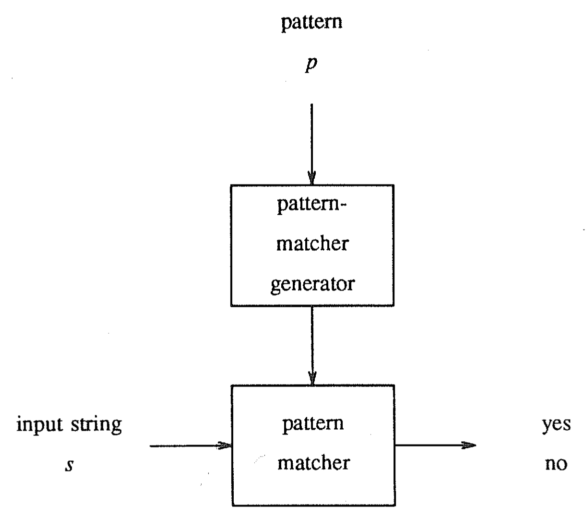

Fig. 1. Model for pattern-matching problems.

We will treat pattern-matching problems in the general setting shown in Fig. 1. The input consists of a pair (p, s) where p is the pattern and s is the input string. The pattern is transformed by the pattern-matcher generator into a pattern matcher, which is used to look for an occurrence of the pattern in the input string. The pattern matcher reports "yes" if s contains a substring matched by p, "no" otherwise.

The actual output of a pattern-matching algorithm depends on the application. In information retrieval, the input string is often a file consisting of lines of text (such as a dictionary, a program listing, or a manuscript), and we are interested in all lines matched by the pattern. For example, we might be interested in searching a dictionary for all words that contain the five vowels in order. In programming language compilation, the input string would be the sequence of characters making up the source program and we would be interested in partitioning the input into a sequence of lexical tokens such as comments, identifiers, operators, and so on, where the structure of the tokens is specified by the pattern. In text editing, we might want to identify the longest nonoverlapping substrings of the input string denoted by the pattern; for example, in a manuscript we might want to change all occurrences of *color* into *colour*. In this paper, we will simply consider the output of the pattern matcher to be "yes" if the input string contains a substring matched by the pattern, "no" otherwise.

We will measure the overall performance of a pattern-matching algorithm by the time and space taken to answer yes or no measured as a function of the lengths of p and s using the random-access machine (RAM) as the model of computation [6]. We will assume the pattern is given before the text string. In this way an algorithm can preprocess the pattern, constructing from it whatever kind of pattern-matching machine it needs before scanning any of the input string. In the setting of Fig. 1, we will analyze the time and space taken by the pattern-matcher generator as well as the pattern matcher itself.

In practice, a number of issues need to be resolved in the design of a patternmatching program. Decisions need to be made on what class of patterns to use, what notation to describe the patterns, and what distribution to expect on patterns and text strings. In some applications it may not be possible to store all of the input string in memory, so there may be a limit on how much backtracking over the input is possible. For applications such as text editing it is desirable to be able to construct the recognizer quickly since the input strings are likely to be short. For applications such as textual search we may be willing to spend more time constructing the recognizer if it can be made to run faster on long text files. In many applications the pattern-matching process is I/O bound, so it is important to be able to read the input string as quickly as possible. In the 1980s the performance of many pattern-matching programs improved by an order of magnitude or more due to algorithmic improvements of the kind discussed in this chapter [73].

2. Notations for patterns

In our pattern-matching model, the pattern matcher reports "yes" if the input string contains a substring matched by the pattern, "no" otherwise. In effect, the pattern is a notation for describing a set of substrings. The simplest patterns are single keywords that match themselves. For example, if we specified *dous* as a pattern, then we would report success on input strings such as *hazardous* and *horrendously*. A somewhat broader class of patterns would be sets of keywords. Since many text-processing systems use variants of regular expressions to describe patterns, we will use regular expressions as they are defined in language theory as our third notation for specifying

patterns. We will also discuss the effect of some of the embellishments that have been added to regular expressions to make them more descriptive for practical use.

2.1. *Regular expressions*

Regular expressions, regular sets, and finite automata are central concepts in automata and formal language theory. As we shall see, these concepts are also central in a study of string pattern-matching algorithms.

DEFINITION. We define *regular expressions* and the strings they match recursively as follows:

(1) The following characters are metacharacters: | ( ) *

(2) A non-metacharacter a is a regular expression that matches the string a.

(3) If r_1 and r_2 are regular expressions, then $(r_1|r_2)$ is a regular expression that matches any string matched by either r_1 or r_2.

(4) If r_1 and r_2 are regular expressions, then $(r_1)(r_2)$ is a regular expression that matches any string of the form xy, where r_1 matches x and r_2 matches y.

(5) If r is a regular expression, then $(r)^*$ is a regular expression that matches any string of the form $x_1x_2 \ldots x_n$, $n \geqslant 0$, where r matches x_i for $1 \leqslant i \leqslant n$. In particular, $(r)^*$ matches the empty string, which we denote by ε.

(6) If r is a regular expression, then (r) is a regular expression that matches the same strings as r.

Many parentheses in regular expressions can be avoided by adopting the convention that the Kleene closure operator * has the highest precedence, then concatenation, then |. The two binary operators, concatenation and |, are left-associative. Under these conventions the regular expressions $(a|((b)^*)(c))$ and $a|b^*c$ are equivalent, in the sense that they match the same strings, namely, an a, or a sequence of zero or more b's followed by a c.

2.1. EXAMPLE. The regular expression

(hot|cold) (apple|blueberry|cherry) (pie|tart)

matches any of the twelve delicacies ranging from *hot apple pie* to *cold cherry tart*. The regular expression

*the (very,)*very hot cherry pie*

matches the strings *the very hot cherry pie*; *the very, very hot cherry pie*; *the very, very, very hot cherry pie*; and so on. The regular expression

$$(aa|bb)^*((ab|ba)(aa|bb)^*(ab|ba)(aa|bb)^*)^*$$

matches all strings of a's and b's having both an even number of a's and an even number of b's.

A *regular set* is a set of strings matched by a regular expression. (Some authors use the terms *rational set* and *rational expression* for regular set and regular expression.) The origins of regular sets go back to the work of McCulloch and Pitts [94] who devised finite-state automata as a model for the behavior of neural nets. As a formalism for specifying strings, regular expressions and finite automata are equivalent in that they both describe the same sets of strings [110]. The notation of regular expressions arises naturally from the mathematical result of Kleene [79] that characterizes the regular sets as the smallest class of sets of strings which contains all finite sets of strings and which is closed under the operations of union, concatenation, and "Kleene closure".

2.2. *Extensions to the regular expression notation*

Many text-editing and searching programs add abbreviations and new operators to the basic regular expression notation above to make it easier to specify patterns. Here we mention some of the extensions used by the popular regular expression matching programs *awk* [7], *egrep* [96], and *lex* [84] on the UNIX® operating system. One useful extension is the quoting metacharacter, \, that permits metacharacters to be matched. In the definition above, the symbols |, (,), and * are metacharacters considered not to be part of the alphabet of input characters. With \, we can include these symbols, as well as \ itself, as targets for matches, by writing \| to match |, * to match *, and so on; \\ matches \.

We often want to match substrings at the beginning or end of the input string. To do this, *awk* adds the metacharacters ^ and $ to match the null string at the left and right ends of an input line. Thus *dous*$ matches the substring *dous* only when it is at the right end of the input line.

Another convenience is a succinct way to specify a match for any character in a set of characters. In *awk*, the symbol . is used to match any single character. We can think of . as a "wild card" or "don't care" symbol. The notation [*abc*] is a character class that matches an *a*, *b*, or *c*, and the notation [^*abc*] is a complemented character class that matches any single character that is not an *a*, *b*, or *c*. Thus the *awk* expression

^[^ *aeiou*]**a* [^ *aeiou*]**e* [^ *aeiou*]**i* [^ *aeiou*]**o* [^ *aeiou*]**u* [^ *aeiou*]*$

will match an input line in which the five vowels appear in lexicographic order, such as a line consisting of the word *abstemious* or *facetious*. A range of characters may be specified with an abbreviated character class; for example, [*a–z*] matches any lower-case letter, [*A–Za–z*] matches any upper- or lower-case letter, and [^0–9] matches any non-digit. None of these extensions, however, extends the descriptive power of regular expressions beyond regular sets.

Many pattern-matching tools allow Boolean combinations of patterns. Since the class of regular sets is closed under the operations of union, intersection, and complement, the Boolean operations *or*, *and*, and *not* do not add more descriptive power to regular expressions as a notation for describing string patterns, but in practice, they offer considerable convenience and, in both theory and practice,

succinctness to the specification of patterns. For example, try writing a regular expression pattern for all words with no repeated letter (such as *ambidextrously* or *dermatoglyphics*). The general question concerning the relative succinctness of different notations for regular sets has been of considerable theoretical interest [65, 99].

2.3. *Regular expressions with back referencing*

In some applications it is desirable to be able to specify repeating structures in matches. For example, we might be interested in words of the form xx, that is, words consisting of a repeated substring such as *killeekillee* or *tangantangan*. Patterns such as these are sometimes called *squares* and they cannot be specified by regular expressions. An assignment operator called *back referencing* can be added to regular expressions to allow repeating patterns to be specified. Back referencing appeared in the first version of the SNOBOL programming language [48] and has been implemented in several commands on the UNIX system, notably the text editor *ed* and the pattern-matching program *grep* [96].

DEFINITION. The following rules define *regular expressions with back referencing* (rewbrs for short) and the strings they match. In these rules we assume the alphabet of characters is distinct from $\{v_1, v_2, \ldots\}$, the set of variable names. In a rewbr, a variable defines a string whose initial value is undefined.

(1) The following characters are metacharacters: | ( ) * %

(2) Each non-metacharacter a is a rewbr that matches the string a.

(3) Each variable v_i is a rewbr that matches the string defined by v_i.

(4) If r_1 and r_2 are rewbrs, then $(r_1|r_2)$ is a rewbr that matches any string matched by either r_1 or r_2.

(5) If r_1 and r_2 are rewbrs, then $(r_1)(r_2)$ is a rewbr that matches any string of the form xy, where r_1 matches x and r_2 matches y.

(6) If r is a rewbr, then $(r)^*$ is a rewbr that matches any string of the form $x_1x_2\ldots x_n$, $n \geqslant 0$, where r matches x_i for $1 \leqslant i \leqslant n$.

(7) If r is a rewbr that matches a string x, then $(r)\%v_i$ is a rewbr that matches x and v_i is assigned the value x.

(8) If r is a rewbr, then (r) is a rewbr that matches the same strings as r.

Rule (7) defines the back-referencing operator %. Its properties are similar to those of the conditional value assignment operator in SNOBOL4 [60]. As before, redundant parentheses can be avoided using the same precedences and associativities as in regular expressions. The back-referencing operator is left-associative and has the highest precedence.

2.2. EXAMPLE. A few examples should clarify this definition.

(1) The rewbr $(a|b|c)^*(a|b|c)\%v_1(a|b|c)^*v_1(a|b|c)^*$ matches any string of a's, b's or c's with at least one repeated character. To see this, note that $(a|b|c)^*$ matches any string of characters and that $(a|b|c)\%v_1$ will match any single character and assign to v_1 the value of that character. The second v_1 in the rewbr will match a second occurrence of that character in the input string.

(2) The rewbr $(a|b|c)^*\%v_1v_1$ matches any string of the form xx where x is any string of a's, b's or c's.

(3) The rewbr $(((a|b|c)^*\%v_1)v_1)^*$ matches any string of the form $x_1x_1x_2x_2\ldots x_nx_n$ where each x_i is a string of a's, b's or c's.

These examples illustrate some of the definitional power of rewbrs. Pattern (1) can be denoted by a somewhat longer ordinary regular expression. The complement of pattern (1) suggests a concise specification for the no-repeated-letter pattern mentioned above. Pattern (2), however, does not denote a regular set or even context-free language so it cannot be denoted by any regular expression or context-free grammar [70]. Pattern (3), likewise, cannot be expressed by a regular expression or context-free grammar. Patterns denoted by regular expressions with back referencing, only one variable name, and no alternation have been studied by Angluin [11].

3. Matching keywords

We are now ready to consider the first of our pattern-matching problems.

3.1. PROBLEM. Given a pattern p consisting of a single keyword and an input string s, answer "yes" if p occurs as a substring of s, that is, if $s=xpy$, for some x and y; "no" otherwise. For convenience, we will assume $p=p_1p_2\ldots p_m$ and $s=s_1s_2\ldots s_n$ where p_i represents the ith character of the pattern and s_j the jth character of the input string.

This section presents and compares four algorithms for this task. It is straightforward to generalize these algorithms to locate all occurrences of the pattern in the input string.

3.1. The brute-force algorithm

Our initial algorithm is the obvious one. First, the pattern-matcher generator reads and stores the pattern; this is its only function. Then, the pattern matcher reads the input string using a buffer, since it may have to backtrack if a partial match fails. It looks for a match by comparing the pattern $p_1p_2\ldots p_m$ with the m-character substring $s_ks_{k+1}\ldots s_{k+m-1}$ for each successive value of k from 1 to $n-m+1$. It compares the keyword with the substring of the input from left to right until it either matches all of the keyword or finds that $p_i\neq s_j$, for some $1\leqslant i\leqslant m$ and $k\leqslant j\leqslant k+m-1$:

$$\begin{array}{ccccccc} & & & i & & & \\ & & & \downarrow & & & \\ & p_1 & \ldots & p_i & \ldots & p_m & \\ s_1\ldots & s_k & \ldots & s_j & \ldots & s_{k+m-1} & \ldots s_n \\ & & & \uparrow & & & \\ & & & j & & & \end{array}$$

```
begin
  $i$:=1 /* pointer into the pattern keyword $p=p_1p_2\ldots p_m$ */
  $j$:=1 /* pointer into the input string $s=s_1s_2\ldots s_n$ */
  while
    $i\leqslant m$ and $j\leqslant n$ do
      if $p_i=s_j$ then
        begin $i$:=$i+1$; $j$:=$j+1$ end
      else
        begin $i$:= 1; $j$:=$j-i+2$ end
  if $i>m$ then return "yes" else return "no"
end
```

Fig. 2. The brute-force algorithm for matching a single keyword.

At this point it "slides" the pattern one character to the right and starts looking for a match by comparing p_1 with s_{k+1}. The pseudo-code in Fig. 2 details this behavior.

3.2. THEOREM. *The brute-force algorithm solves Problem* 3.1 *in* O(mn) *time and* O(m) *space.*

PROOF. The algorithm may take the maximum O(mn) time. For example, when $p=a^{m-1}b$ and $x=a^n$, the algorithm makes $mn-m^2+m-1$ character comparisons. The algorithm requires a buffer of size O(m) to hold the pattern and the m-character substring of the input. □

In practical situations the expected performance of the brute-force algorithm is usually O($m+n$), but a precise characterization depends on the statistical properties of the pattern and text string.

3.2. The Karp–Rabin algorithm

The brute-force algorithm looks for a match by comparing the pattern with the m-character substring $s_ks_{k+1}\ldots s_{k+m-1}$ for each successive value of k from 1 to $n-m+1$. Karp and Rabin [78] suggested that we use a judiciously designed hash function h, which they call a *fingerprint*, to lower the cost of comparing the pattern with each successive m-character substring.

Let h be a hash function that maps each m-character string to an integer. If $h(p_1p_2\ldots p_m)\neq h(s_ks_{k+1}\ldots s_{k+m-1})$, then we know $p_1p_2\ldots p_m$ cannot possibly match $s_ks_{k+1}\ldots s_{k+m-1}$. If, however, $h(p_1p_2\ldots p_m)=h(s_ks_{k+1}\ldots s_{k+m-1})$, then we must still compare $p_1p_2\ldots p_m$ with $s_ks_{k+1}\ldots s_{k+m-1}$ character by character to make sure we do not have a false match.

The hash function reduces m character comparisons into a single integer comparison. But we have not gained very much if it takes a long time to compute the hash value or if we get a lot of false matches. Karp and Rabin have suggested using the hash function $h(x)=x \bmod q$ where q is an appropriately large prime. We can transform the

m-character string $s_k s_{k+1} \ldots s_{k+m-1}$ into the integer

$$x_k = s_k b^{m-1} + s_{k+1} b^{m-2} + \cdots + s_{k+m-1}$$

by treating each character s_i as an integer and choosing b to be an appropriate radix. The value of x_{k+1} can be simply computed from the value of x_k:

$$x_{k+1} = (x_k - s_k b^{m-1})b + s_{k+m}.$$

Figure 3 contains the details of the Karp–Rabin algorithm as presented by Sedgewick [120].

```
begin
  $d := b^{m-1} \bmod q$
  $h_p := (p_1 * b^{m-1} + p_2 * b^{m-2} + \cdots + p_m) \bmod q$
  $h_s := (s_1 * b^{m-1} + s_2 * b^{m-2} + \cdots + s_m) \bmod q$
  for $k := 1$ to $n-m+1$ do
  begin
    if $h_p = h_s$ and $p_1 p_2 \ldots p_m = s_k s_{k+1} \ldots s_{k+m-1}$ then return "yes"
    $h_s := (h_s + b*q - s_k * d) \bmod q$ /* the additive factor $b*q$ is to keep the rhs positive */
    $h_s := (h_s * b + s_{k+m}) \bmod q$
    $k := k+1$
  end
  return "no"
end
```

Fig. 3. The Karp–Rabin algorithm for matching a single keyword.

Since the mod function is associative, we can apply it after each arithmetic operation to keep the numbers small and we will still end up with the same result as though we had performed all of the operations first and then applied the mod function.

In the worst case, at each iteration we might get a false match that results in m character comparisons to determine $p_1 p_2 \ldots p_k \neq s_k s_{k+1} \ldots s_{k+m-1}$. Most of the time, however, we would expect $h_p \neq h_s$ and thus each iteration would be done in constant time.

3.3. THEOREM. *The Karp–Rabin algorithm solves Problem* 3.1 *in* O(mn) *time in the worst case and in* O($m+n$) *time in the expected case. It requires* O(m) *space.*

One other advantage of the Karp–Rabin algorithm is that it can be used to do two-dimensional pattern matching [78].

3.3. The Knuth–Morris–Pratt algorithm

When a mismatch occurs at the jth input character in the brute-force algorithm, we do not need to reset the input pointer to position $j-i+2$ because the previous $i-1$

input characters are already known—they are the first $i-1$ characters of the pattern. Taking advantage of this observation, Knuth, Morris, and Pratt [80] published an elegant nonbacktracking algorithm that requires only O($m+n$) time in the worst case to determine whether p is a substring of s. The algorithm first reads the pattern and in O(m) time constructs a table h, called the *next* function, that determines how many characters to slide the pattern to the right in case of a mismatch during the pattern-matching process. The pattern matcher then processes the input string. Since no backtracking is required, the input can be read one character at a time. The pattern matcher uses the table h in the way shown in Fig. 4 to locate a match.

```
begin
  $i := 1$ /* pointer into the pattern $p = p_1 p_2 \ldots p_m$ */
  $j := 1$ /* pointer into the input string $s = s_1 s_2 \ldots s_n$ */
  while $i \leq m$ and $j \leq n$ do
  begin
    while $i > 0$ cand $p_i \neq s_j$ do $i := h_i$
    $i := i+1$; $j := j+1$
  end
  if $i > m$ then return "yes" else return "no"
end
```

Fig. 4. The Knuth–Morris–Pratt algorithm for matching a single keyword.

The **cand** in the inner loop is a "conditional and" that compares p_i and s_j only if $i > 0$. The key idea is that if we have successfully matched the prefix $p_1 p_2 \ldots p_{i-1}$ of the keyword with the substring $s_{j-i+1} s_{j-i+2} \ldots s_{j-1}$ of the input string and $p_i \neq s_j$, then we do not need to reprocess any of $s_{j-i+1} s_{j-i+2} \ldots s_{j-1}$ since we know this portion of the text string is the prefix of the keyword that we have just matched.

Instead, each iteration of the inner **while**-loop slides the pattern a certain number of characters to the right as determined by the next table h. In particular, the pattern is shifted $i - h_i$ positions to the right and i is set to h_i. The algorithm repeats this step until i becomes zero (in which case none of the pattern matches any of substring of the input string ending at character s_j) or until $p_i = s_j$ (in which case $p_1 \ldots p_{i-1} p_i$ matches $s_{j-i+1} \ldots s_{j-1} s_j$ for the new value of i). The outer **while**-loop increments the pointers to the pattern and the input string.

The essence of the algorithm is the next function that is stored in the table h. To make the algorithm run correctly and in linear time it has the property that h_i is the largest k less than i such that $p_1 p_2 \ldots p_{k-1}$ is a suffix of $p_1 p_2 \ldots p_{i-1}$ (i.e., $p_1 \ldots p_{k-1} = p_{i-k+1} \ldots p_{i-1}$) and $p_i \neq p_k$. If there is no such k, then $h_i = 0$. One easy way to compute h is by the program in Fig. 5 that is virtually identical to the matching program itself.

The assignment statement $j := h_j$ in the inner loop (the second **while**-statement in Fig. 5) is never executed more often than the statement $i := i+1$ in the outer loop. Therefore, the computation of h is done in O(m) time. Similarly, the assignment $i := h_i$ in the inner

```
begin
  $i:=1$; $j:=0$; $h_1:=0$
  while $i<m$ do
  begin
    while $j>0$ cand $p_i \neq p_j$ do $j:=h_j$
    $i:=i+1$; $j:=j+1$
    if $p_i=p_j$ then $h_i:=h_j$ else $h_i:=j$
  end
end
```

Fig. 5. Algorithm to compute the next function h for pattern $p=p_1p_2\cdots p_m$.

loop of Fig. 4 is never executed more often than the statement $j:=j+1$ in the outer loop. What this means is that the pattern is shifted to the right a total of at most n times by the inner loop. Therefore, the pattern matcher runs in $O(n)$ time. Consequently, we have the following theorem.

3.4. THEOREM. *The Knuth–Morris–Pratt algorithm takes* $O(m+n)$ *time and* $O(m)$ *space to solve Problem* 3.1.

Note that the running time of the Knuth–Morris–Pratt algorithm is independent of the size of alphabet. As another measure of performance, let us briefly examine how many times the inner loop of the Knuth–Morris–Pratt algorithm can be executed while scanning the same input character.

DEFINITION. The Fibonacci strings are defined as follows:

$$F_1=b, \qquad F_2=a, \quad \text{and} \quad F_n=F_{n-1}F_{n-2} \quad \text{for } n>2.$$

3.5. EXAMPLE. Consider the Fibonacci string $F_7=abaababaabaab$. The algorithm in Fig. 5 produces the following next function for F_7:

i	1	2	3	4	5	6	7	8	9	10	11	12	13
p_i	a	b	a	a	b	a	b	a	a	b	a	a	b
h_i	0	1	0	2	1	0	4	0	2	1	0	7	1

Let us apply the Knuth–Morris–Pratt algorithm to look for the pattern F_7 in the input string *abaababaabacabaababaabaab*. After matching the first 11 characters successfully, the algorithm finds a mismatch at the input character c. At this point $i=j=12$:

```
                      i
                      ↓
a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b
                      ↑
                      j
```

The first iteration of the inner loop of Fig. 4 sets $i=h_{12}=7$. This has the effect of shifting the pattern 5 characters to the right, so position 7 of the pattern is now aligned above position 12 of the input string:

```
                      i
                      ↓
          a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b
                      ↑
                      j
```

At this point, p_i still does not match s_j, so i is set to $h_7=4$. Mismatches continue for $i=4,2,1,0$. At this point the inner loop is exhausted without finding a match and the outer loop then sets i to 1 and j to 13:

```
                        i
                        ↓
                        a b a a b a b a a b a a b
a b a a b a b a a b a c a b a a b a b a a b a a b
                        ↑
                        j
```

From this point, the algorithm finds a successful match.

Matching a Fibonacci string of length m presents a worst case for the Knuth–Morris–Pratt algorithm. In one execution of the inner loop of Fig. 4 the pattern may be shifted $\log_\varphi m$ times, where $\varphi=\frac{1}{2}(1+\sqrt{5})$, the golden ratio. This is the maximum that is possible [80].

3.6. THEOREM. *The maximum number of times the inner loop of the Knuth–Morris–Pratt algorithm can shift the pattern right while scanning the same input character is at most* $1+\log_\varphi m$.

Note, however, that the total number of shifts of the pattern made by the algorithm in processing the complete input string is at most $n-1$.

3.4. The Boyer–Moore algorithm

The Boyer–Moore [32] algorithm for string matching is the fastest-known single-keyword pattern-matching algorithm in both theory and practice. It achieves its great speed by skipping over portions of the input string that cannot possibly contribute to a match. When the alphabet size is large, the algorithm determines whether a match occurs comparing only about n/m input string characters on the average.

The basic idea of the algorithm is to superpose the keyword on top of the input string and to look for a match by comparing characters in the keyword from right to left against the input string. Consequently, like the brute-force algorithm, it needs a buffer in which to store the current portion of the input. Initially, we compare p_m with s_m. If s_m occurs nowhere in the keyword, then there cannot be a match for the pattern beginning at any of the first m characters of the input. We can therefore safely slide the keyword m characters to the right and try matching p_m with s_{2m}, thus avoiding $m-1$ unnecessary character comparisons.

Consider the general case. We have just shifted the pattern to the right and are about to compare p_m with s_k:

$$\begin{array}{lll} & & \downarrow \\ p_1 & & \cdots p_m \\ s_1 \cdots s_{k-m+1} & \cdots & s_k \cdots s_n \\ & & \uparrow \\ & & j \end{array}$$

(1) We discover that p_m and s_k do not match. If the rightmost occurrence of s_k in the keyword is p_{m-g}, we can shift the pattern g positions to the right to align p_{m-g} and s_k, and then resume matching by comparing p_m with s_{k+g}:

$$\begin{array}{llll} & & & \downarrow \\ p_1 & & \cdots p_{m-g} \cdots & p_m \\ s_1 \cdots s_{k-m+g+1} \cdots & s_k & \cdots & s_{k+g} \cdots s_n \\ & & & \uparrow \\ & & & j \end{array}$$

As a special case, if s_k did not occur in the keyword, we would shift the pattern m positions to the right and resume matching by comparing p_m with s_{k+m}.

(2) Suppose the last $m-i$ characters of the keyword agree with the last $m-i$ characters of input string ending at position k; that is,

$$p_{i+1}p_{i+2}\cdots p_m = s_{k-m+i+1}s_{k-m+i+2}\cdots s_k.$$

If $i=0$, we have found a match. On the other hand, suppose $i>0$ and $p_i \neq s_{k-m+i}$. Two cases now arise.

(a) If the rightmost occurrence of the character s_{k-m+i} in the keyword is p_{i-g}, then as in Case 1 we can simply shift the keyword g positions to the right so that p_{i-g} and s_{k-m+i} are aligned and resume matching by comparing p_m with s_{k+g}:

$$\begin{array}{llll} & & & \downarrow \\ p_1 & \cdots p_{i-g} & & \cdots p_m \\ s_1 \cdots s_{k-m+g+1} \cdots & s_{k-m+i} \cdots & & s_{k+g} \cdots s_n \\ & & & \uparrow \\ & & & j \end{array}$$

If p_{i-g} is to the right of p_i, i.e., $g<0$, then we would instead shift the pattern one position to the right and resume matching by comparing p_m with s_{k+1}.

(b) A longer shift than that obtained in Case 2(a) may be possible. Suppose the suffix $p_{i+1}p_{i+2}\cdots p_m$ reoccurs as the substring $p_{i+1-g}p_{i+2-g}\cdots p_{m-g}$ in the keyword and $p_i \neq p_{i-g}$. (If there is more than one such reoccurrence, take the rightmost one.) Then we may get more of a shift than in Case 2(a) by aligning $p_{i+1-g}p_{i+2-g}\cdots p_{m-g}$ above $s_{k-m+i+1}s_{k-m+i+2}\cdots s_k$ and recommencing the search by comparing p_m with s_{k+g}:

$$\begin{array}{lllll} & & & & \downarrow \\ p_1 & \cdots p_{i+1-g} & \cdots p_{m-g} \cdots & & p_m \\ s_1 \cdots s_{k-m+g+1} \cdots & s_{k-m+i+1} \cdots & s_k & \cdots & s_{k+g} \cdots s_n \\ & & & & \uparrow \\ & & & & j \end{array}$$

The details of this process are given in Fig. 6.

```
begin
  j:=m
  while $j \leqslant n$ do
  begin
    i:=m
    while $i>0$ cand $p_i = s_j$ do
      begin $i:=i-1$; $j:=j-1$ end
    if $i=0$ then return "yes"
    else $j:=j+\max(d_1[s_j], d_2[i])$
  end
  return "no"
end
```

Fig. 6. The Boyer–Moore algorithm for matching a single keyword.

This algorithm uses two tables d_1 and d_2 to determine how far to slide the keyword to the right when $p_i \neq s_j$. The first table is indexed by characters. For every character c, $d_1[c]$ is the largest i such that $c=p_i$, or $c=m$ if the character c does not occur in the keyword. Table d_1 covers Cases 1 and 2(a) in the discussion above.

Case 2(b) is handled by the second table, which is indexed by positions in the keyword. For every $1 \leqslant i \leqslant m$, $d_2[i]$ gives the minimum shift g such that when we align p_m above s_{k+g}, the substring $p_{i+1-g}p_{i+2-g}\cdots p_{m-g}$ of the pattern agrees with the substring $s_{k-m+i+1}s_{k-m+i+2}\cdots s_k$ of input string, assuming p_i did not match s_{k-m+i}. Formally,

$$\begin{aligned} d_2[i] = \min\{g+m-i \mid g \geqslant 1 \text{ and } (g \geqslant i \text{ or } p_{i-g} \neq p_i) \\ \text{and } ((g \geqslant k \text{ or } p_{k-g}=p_k) \text{ for } i<k\leqslant m)\}. \end{aligned}$$

This table can be computed using the algorithm in Fig. 7.

```
begin
  for $i := 1$ to $m$ do $d_2[i] := 2*m - i$
  $j := m$; $k := m + 1$
  while $j > 0$ do
  begin
    $f[j] := k$
    while $k \leqslant m$ cand $p_j \neq p_k$ do
    begin
      $d_2[k] := \min(d_2[k], m - j)$
      $k := f[k]$
    end
    $j := j - 1$; $k := k - 1$
  end
  for $i := 1$ to $k$ do $d_2[i] := \min(d_2[i], m + k - i)$
  $j := f[k]$
  while $k \leqslant m$ do
  begin
    while $k \leqslant j$ do
    begin
      $d_2[k] := \min(d_2[k], j - k + m)$
      $k := k + 1$
    end
    $j := f[j]$
  end
end
```

Fig. 7. Algorithm to compute shift table d_2.

Boyer and Moore originally had a different way of computing the second shift table d_2. The technique given above is due to Knuth [80] with a modification provided by Mehlhorn [124]. The intermediate function f computed by the algorithm in Fig. 7 has the property that $f[m] = m + 1$ and for $1 \leqslant j < m$, $f[j] = \min\{i \mid j < i \leqslant m$ and $p_{i+1}p_{i+2} \ldots p_m = p_{j+1}p_{j+2} \ldots p_{m+j-i}\}$.

3.7. EXAMPLE. The algorithm in Fig. 7 produces the following values for d_2 and f for the pattern *abaababaabaab*:

i	1	2	3	4	5	6	7	8	9	10	11	12	13
p_i	*a*	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*
$d_2[i]$	20	19	18	17	16	15	14	8	15	14	8	14	1
$f[i]$	9	10	11	12	8	9	10	11	12	13	13	13	14

The minimum number of character comparisons needed to determine all occurrences of a keyword of length m in an input string of length n is an interesting theoretical question that has been considered by Knuth [80], Guibas and Odlyzko [61], and Galil [53]. The most recent work by Apostolico and Giancarlo [18] has resulted in a variant of the Boyer–Moore algorithm in which the number of character comparisons is at most $2n$, regardless of the number of occurrences of the pattern in the input string. The following theorem summarizes the worst-case behavior of the Boyer–Moore algorithm.

3.8. THEOREM. *The Boyer–Moore algorithm solves Problem* 3.1 *in* $O(m+n)$ *time and* $O(m)$ *space.*

Rivest [114] has shown that any algorithm for finding a keyword in an input string must examine at least $n - m + 1$ of the characters in the input string in the worst case. This result implies that there do not exist pattern-matching algorithms whose worst-case behavior is sublinear in n, providing a sharp contrast with the sublinear average behavior of the Boyer–Moore algorithm. Yao [139] has shown that the minimum average number of characters that need to be examined in looking for a pattern in a random text string is $\Omega(n\lceil\log_A m\rceil/m)$ for $n > 2m$, where A is the alphabet size.

3.5. *Expected performance*

Problem 3.1 has been one of the most intensely studied string-matching questions in both theory and practice. The quintessential question is how well does an algorithm solve Problem 3.1 in practice. Several authors have conducted experiments evaluating the relative performance of the four algorithms presented in this section [44, 71, 124]. The primary conclusion is that the Boyer–Moore algorithm is noticeably faster for longer patterns especially in text-processing applications. Horspool observed that for pattern lengths of six or more the Boyer–Moore algorithm even outperformed a naive algorithm that was implemented using a special-purpose machine instruction to scan the input string for the character with the lowest frequency in the pattern [71]. He also noted that the performance of the simplified form of the algorithm in Fig. 8 with a single shift table indexed by characters is comparable to the original formulation. The comparison of $s_{j-m+1}s_{j-m+2} \ldots s_j$ with $p_1p_2 \ldots p_m$ is done from right to left.

Several authors have used Markov-chain theory to derive analytical results on the

```
begin
  for each character $c$ in the input alphabet do $d[c] := m$
  for $j := 1$ to $m - 1$ do $d[p_j] := m - j$
  $j := m$
  while $j \leqslant n$ do
  begin
    if $s_j = p_m$ then
      if $s_{j-m+1}s_{j-m+2} \ldots s_j = p_1p_2 \ldots p_m$ then return "yes"
    $j := j + d[s_j]$
  end
  return "no"
end
```

Fig. 8. Horspool's simplified Boyer–Moore algorithm.

expected number of character comparisons made by these algorithms on random strings [21, 23]. Baeza-Yates [21] and Schaback [119] have analyzed the expected performance of some variants of the Boyer–Moore algorithm.

3.6. *Theoretical considerations*

In 1971, Cook [38] published the following surprising result that has had a significant influence on the theory of string pattern matching.

3.9. THEOREM. *Every two-way deterministic pushdown automaton (2DPDA) language can be recognized in linear time on a random-access machine.*

This result states that there exists a linear-time pattern-matching algorithm for any set of strings that can be recognized by a 2DPDA, even though the 2DPDA may spend more than linear time recognizing the set of strings. For example, the existence of an $O(m+n)$ keyword-matching algorithm follows directly from Cook's result because the set of strings

$$\{p\#s \mid s=xpy \text{ for some } x \text{ and } y\}$$

can be recognized by a 2DPDA, and hence can be recognized in linear time on a random-access machine [6, 98]. The string-matching capabilities of other classes of automata, especially k-head finite automata, have also been of theoretical interest [16, 39, 58, 85].

Knuth, Morris, and Pratt give a fascinating account about the influence of Cook's theoretical result on their algorithm [80]. Knuth traced out the simulation used in Cook's constructive proof to derive a linear-time pattern-matching algorithm for the keyword-recognition problem. Pratt modified this algorithm to make its running time independent of the input alphabet size. The resulting algorithm was one which Morris had discovered and implemented independently, without the benefit of Cook's theorem.

The string-matching problem has added new vigor to the study of periods and overlaps in strings and to the study of the combinatorics of patterns in strings [24, 43, 46, 62, 63, 80]. We say that an integer k is a *period* of $s=s_1s_2\ldots s_n$ if $s_i=s_{i+k}$ for $1\leqslant i\leqslant n-k$. The following two statements are equivalent:

(1) k is a period of s if and only if $s=(uv)^ju$ for some $j\geqslant 0$, where $|uv|=k$ and v is nonempty ($|x|$ denotes the length of a string x).

(2) k is a period of s if and only if $st=rs$ for some equal-length strings r and t.

When there is a mismatch in the Knuth–Morris–Pratt algorithm at symbol p_i in the pattern, the length of the ensuing shift is a period of $p_1\ldots p_{i-1}$. In general, $h_i=i-k$, where k is the smallest period of $p_1\ldots p_{i-1}$ that is not a period of $p_1\ldots p_i$. Note that the Knuth–Morris–Pratt algorithm can be used to compute the period of a string in linear time.

An important mathematical property of periods is that if i and j are periods of s, and $i+j\leqslant|s|+gcd(i,j)$, then the $gcd(i,j)$ is also a period of s [88].

Galil and Seiferas [58] and Crochemore and Perrin [43] have developed keyword-matching algorithms that use only a constant amount of additional space and are intermediate in performance between the Knuth–Morris–Pratt and Boyer–Moore algorithms.

4. Matching sets of keywords

We now generalize the single-keyword pattern-matching problem to sets of keywords.

4.1. PROBLEM. Given a pattern p consisting of a set of keywords $\{w_1, w_2, \ldots, w_k\}$ and an input string $s=s_1s_2\ldots s_n$, answer "yes" if some keyword w_i occurs as a substring of s; "no" otherwise. We assume the sum of the lengths of the keywords is m.

In this section we describe two algorithms to solve this problem. The first constructs an automaton from the keywords to look for the matches in parallel rather than one at a time. The second adds Boyer–Moore-like techniques to the first algorithm. Both algorithms can also be used to locate all occurrences of the keywords in the input string.

4.1. *The Aho–Corasick algorithm*

The straightforward way to look for multiple keywords in an input string is to apply the fastest single-keyword pattern-matching algorithm once for each keyword. This would give a solution whose running time is $O(m+kn)$. In this section we describe an algorithm due to Aho and Corasick [3] that solves this problem in $O(m+n)$ time, which yields a significant improvement when the number of keywords is large.

As for a single keyword, the existence of an $O(m+n)$ time algorithm for the multiple-keyword pattern-matching problem is evident from Cook's Theorem. The language

$$L=\{w_1\#w_2\#\ldots\#w_k\#\#s \mid s=xw_iy \text{ for some } x \text{ and } y, \text{ and } 1\leqslant i\leqslant k\}$$

can be recognized by a 2DPDA, and consequently Cook's result implies that there is an $O(m+n)$ algorithm to do the matching.

The Aho–Corasick algorithm generalizes the Knuth–Morris–Pratt algorithm to multiple-keyword patterns. It first constructs from the set of keywords a pattern-matching automaton in $O(m)$ time; the automaton then reads the input and looks for all the keywords simultaneously in $O(n)$ time. The pattern-matching automaton is like a deterministic finite automaton except that it has two transition functions, a forward transition function and a failure transition function. The failure transition function is used only when the forward transition fails.

DEFINITION. An Aho–Corasick pattern-matching automaton consists of the following components:

(1) Q, a finite set of states,
(2) Σ, a finite input alphabet,
(3) $g: Q \times \Sigma \rightarrow Q \cup \{fail\}$, a forward transition function,
(4) $h: Q \rightarrow Q$, a failure transition function,
(5) q_0, an initial state, and
(6) F, a set of accepting states.

Figure 9 shows how this automaton is used to determine whether an input string $s = s_1 s_2 \ldots s_n$ contains a keyword from the pattern set.

```
begin
  $q := q_0$
  for $j := 1$ to $n$ do
  begin
    while $g[q, s_j] = fail$ do $q := h[q]$
    $q := g[q, s_j]$
    if $q$ is in $F$ then return "yes"
  end
  return "no"
end
```

Fig. 9. The Aho–Corasick algorithm for matching multiple keywords.

Initially, the pattern-matching automaton is in state q_0 scanning s_1, the first character of s. It then executes a sequence of moves. During a move on input symbol s_j the automaton makes zero or more failure transitions until it reaches a state q for which $g[q, s_j] \neq fail$. To complete the move, the automaton makes one forward transition to state $g[q, s_j]$. If this state is an accepting state, the automaton returns "yes" and halts; otherwise, the automaton makes a move on the next input character s_{j+1}.

The forward and failure transition functions of the pattern-matching automaton will have the following two properties:

(1) $g[q_0, a] \neq fail$ for all a in Σ.

(2) If $h[q] = q'$, then the depth of q' is less than the depth of q, where the *depth* of a state is the length of the shortest sequence of forward transitions from the initial state to that state.

The first property makes sure no failure transitions occur in the initial state. The second property ensures that the total number of failure transitions needed to process an input string will be less than the total number of forward transitions. Since exactly one forward transition is made on each input character, fewer than $2n$ transitions of both kinds will be made in processing an input string of length n. Thus, an Aho–Corasick pattern-matching automaton runs in $O(n)$ time.

We now show how to construct the automaton from the set of keywords $p = \{w_1, w_2, \ldots, w_k\}$.

(1) From the set of keywords, construct a trie in which each node represents a prefix of some keyword. The trie can be constructed in $O(m)$ time. The nodes of the trie are the states of the automaton and the root is the initial state q_0, which represents the empty prefix. Each node corresponding to a complete keyword is an accepting state. The transition function g is defined so that $g[q, p_i] = q'$ when q corresponds to the prefix $p_1 \ldots p_{i-1}$ of some keyword and q' corresponds to $p_1 \ldots p_{i-1} p_i$.

(2) For state q_0, set $g[q_0, a] = q_0$ for each character a for which $g[q_0, a]$ was not defined in step (1).

(3) Set $g[q, a] = fail$ for all q and a for which $g[q, a]$ was not defined in steps (1) and (2).

The three steps above define the forward transition function. Note that state q_0 has the property that $g[q_0, a] \neq fail$ for any a in Σ.

DEFINITION. We define a *failure function* $f: Q \rightarrow Q$ on the nodes of the trie with the following property:

> if states q_u and q_v represent the prefixes u and v of some keywords in p, then $f[q_u] = q_v$ if and only if v is the longest proper suffix of u that is also the prefix of some keyword in p.

The failure function can be computed in $O(m)$ time by making a traversal over the trie. We use a breadth-first traversal so that all states of depth d are visited before those of depth $d+1$. During the traversal, we compute the failure function at each state as follows:

(1) Set $f[q_0] = q_0$ and for all states q of depth 1 set $f[q] = q_0$.

(2) Assume that f has been defined for all states of depth less than $d \geqslant 2$ and suppose $g[q, a] = q'$ where q' is a state of depth d. Set $f[q'] = g[r, a]$, where r is the state determined by the following program:

```
$r := f[q]$
while $g[r, a] = fail$ do $r \leftarrow f[r]$
```

Note that since the depth of $f[r]$ is always less than that of r for all states r except the root, and $g[q_0, a] \neq fail$, the **while**-loop will always terminate.

4.2. EXAMPLE. Figure 10 gives the trie for the Fibonacci string *abaababa* and the values of the failure function at each state.

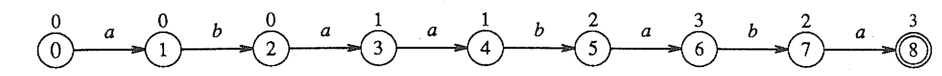

Fig. 10.

The failure function itself can be used as the failure transition function. However, the failure function may sometimes make more failure transitions than necessary. For example, if we try to match the input string *abaabb* with the failure function above, the pattern-matching automaton would make the following state transitions:

```
    a   b   a   a   b   b
  0   1   2   3   4   5   0
                      2
                      0
```

On each of the first five characters, the automaton makes a single forward transition. On the last character, the automaton makes a failure transition from state 5 to state 2, and then another failure transition to state 0, before making the final forward transition to state 0.

An "optimized" failure transition function h that avoids these unnecessary failure transitions can be constructed from f as follows:

(1) Let $h[q_0]=q_0$.

(2) Assume that h has been defined for all states of depth less than d and let q be a state of depth d. If the set of characters for which there is a forward non-*fail* transition in state $f[q]$ is a subset of the set of characters for which there is a forward non-*fail* transition in state q, then set $h[q]=h[f[q]]$; otherwise, set $h[q]=f[q]$.

The function h here is a generalization of the next function in the Knuth–Morris–Pratt algorithm. It can be computed in O(m) time by making another breadth-first traversal over the trie.

4.3. Example. Figure 11 shows the h function for the Fibonacci string *abaababa*.

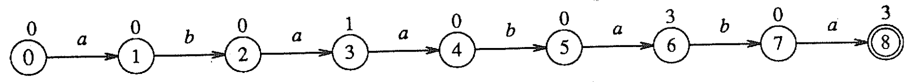

Fig. 11.

With h as the failure transition function, the automaton would make the following moves:

```
    a   b   a   a   b   b
  0   1   2   3   4   5   0
                      0
```

At the last character, this automaton makes one failure transition rather than two.

4.4. Example. For a more complete example, let us construct the Aho–Corasick pattern-matching automaton for the set of keywords {*cacbaa, acb, aba, acbab, ccbab*}. Figure 12 shows the trie. The forward transition function g, the failure function f, and the failure transition function h for this set of patterns are given in Table 1.

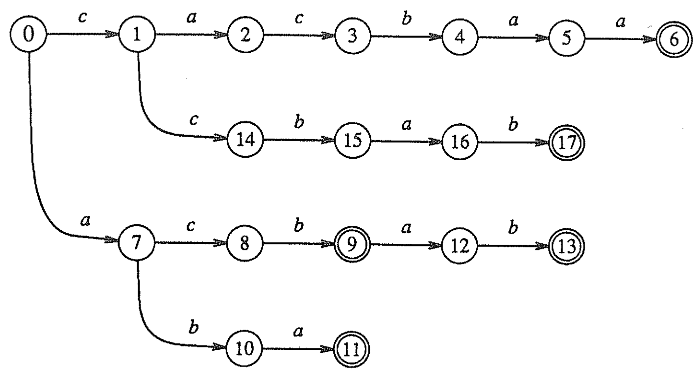

Fig. 12. Trie for keywords *cacbaa, acb, aba, acbab, ccbab*.

Table 1
Transition functions for the pattern-matching automaton of Example 4.5

	g			*f*	*h*
	a	*b*	*c*		
0	7	0	1	0	0
1	2	*fail*	14	0	0
2	*fail*	*fail*	3	7	7
3	*fail*	4	*fail*	8	1
4	5	*fail*	*fail*	9	0
5	6	*fail*	*fail*	12	12
6	*fail*	*fail*	*fail*	7	7
7	*fail*	10	8	0	0
8	*fail*	9	*fail*	1	1
9	12	*fail*	*fail*	0	0
10	11	*fail*	*fail*	0	0
11	*fail*	*fail*	*fail*	7	7
12	*fail*	13	*fail*	7	7
13	*fail*	*fail*	*fail*	10	10
14	*fail*	15	*fail*	1	1
15	16	*fail*	*fail*	0	0
16	*fail*	17	*fail*	7	7
17	*fail*	*fail*	*fail*	10	10

To summarize, we can construct the trie, and the forward and failure transition functions in O(m) time and O(m) space. The pattern-matching automaton processes the input string with no backtracking, making n forward transitions and at most $n-1$ failure transitions on input string of length n. Thus, the pattern matcher runs in O(n) time.

4.5. Theorem. *The Aho–Corasick algorithm solves Problem* 4.1 *in* O($m+n$) *time and* O(m) *space.*

Let us examine the number of failure transitions that can be made while scanning one input character. If the pattern consists of only one keyword of length m, then the Aho–Corasick algorithm is essentially the Knuth–Morris–Pratt algorithm, and as we noted, $\log_\varphi m$ failure transitions are necessary and sufficient in any one move. If the pattern is a set of keywords, then $O(m)$ failure transitions may be necessary in a single move, where m is the sum of the lengths of the keywords. The total number of failure transitions in processing an input string of length n, however, is at most $n-1$.

In the early 1970s Aho and Corasick used this algorithm in a bibliographic search system in which a user could specify documents by prescribing Boolean combinations of keywords and phrases. When this algorithm was used in place of a straightforward multiple-keyword algorithm (the analog of the brute-force method for single keywords), the system typically ran 4 to 12 times faster [3]. Aho also implemented the original version of the UNIX system keyword-matching program *fgrep* using this algorithm.

The Aho–Corasick algorithm can also be used to match subtrees in trees by noting that if we label the branches of each node with the numbers 1, 2, 3, ..., from left to right, then a tree is uniquely characterized by the set of paths from the root to the leaves [4, 69]. Ben-Yehuda and Pinter [25] have extended the Aho–Corasick algorithm to do two-dimensional pattern matching.

4.2. *The Commentz-Walter algorithm*

Commentz-Walter has described an approach in which the ideas in the Boyer–Moore algorithm are combined with the Aho–Corasick algorithm to look for patterns consisting of sets of keywords [37]. The basic idea is to construct an Aho–Corasick style pattern-matching automaton, but for the keywords reversed. Let $k_{\min}$ be the length of a shortest keyword. Matching begins with the automaton in its initial state scanning s_i, the ith character of the input string, where $i=k_{\min}$. We can think of the automaton being superposed on top of the input string with the start state above the character s_i. The automaton then makes state transitions, reading the input string from right to left, until it either finds a match for a keyword or enters a state for which there is no transition on the current input symbol.

In the latter case, the automaton has read the input characters $s_{i-j+1}s_{i-j+2}\ldots s_i$ (from right to left), the automaton is in some state q, and there is no transition from state q on s_{i-j}. At this point, the automaton shifts its start state to a character to the right of s_i and recommences matching from the start state. The amount of the shift is determined as follows:

(1) Let $d_1[q]$ be a minimal shift so that $s_{i-j+1}s_{i-j+2}\ldots s_i$ will be aligned with a matching substring of some keyword.

(2) Let $d_2[q]$ be a minimal shift so that a suffix of $s_{i-j+1}s_{i-j+2}\ldots s_i$ will be aligned with a prefix of some keyword.

(3) Let $d_3[s_{i-j}, j]$ be a minimal shift so that s_{i-j} will be aligned with a matching character in some keyword.

The amount of shift at state q on input character s_{i-j} is then given by

$$shift[q, s_{i-j}, j] = \min\begin{cases}\max(d_1[q], d_3[s_{i-j}, j]),\\ d_2[q].\end{cases}$$

We can visualize each shift as moving the start state of the automaton to the right along the input string.

We will now describe how to construct the Commentz-Walter pattern-matching automaton from the set of keywords. We first construct a trie for the set of keywords reversed. As in the Aho–Corasick automaton, the states are the nodes of the trie, the root is the start state, and each node corresponding to a complete keyword in reverse is an accepting state. Let *path*(q) be the string spelled out by the characters on the path from the start state q_0 to the node q. The transition function g is defined so that $g[q, p_i]=q'$ where $path(q)=p_1p_2\ldots p_{i-1}$ for some reversed keyword $p_1p_2\ldots p_k$, $k\geq i$.

4.6. EXAMPLE. The Commentz-Walter trie for the set of keywords {*cacbaa*, *aba*, *acbab*, *ccbab*, *acb*} is shown in Fig. 13.

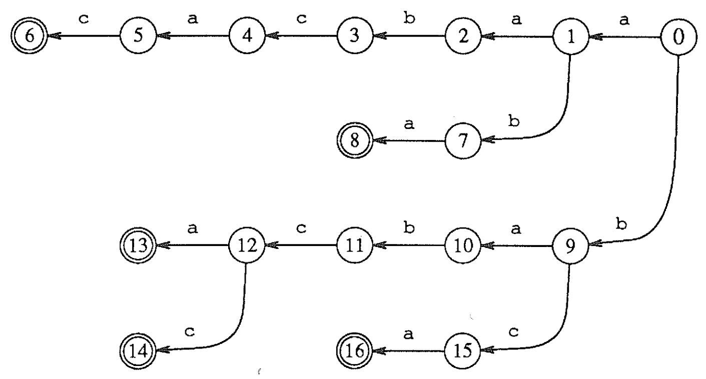

Fig. 13. Reversed trie for keywords *cacbaa*, *aba*, *acbab*, *ccbab*, *acb*.

We will now compute the two intermediate shift functions d_1 and d_2 for each state. Let *depth*(q) be the number of edges along the path from the root to node q. We compute two intermediate sets for each state other than the start state:

$$set_1(q)=\{r \mid path(q) \text{ is a proper suffix of } path(r)\},$$
$$set_2(q)=\{r \mid r \text{ is in } set_1(q) \text{ and } r \text{ is an accepting state}\}.$$

From these sets we compute the two shift functions d_1 and d_2. For the start state q_0, we define $d_1[q_0]=1$. For all other states q, we define

$$d_1[q]=\min\begin{cases} depth(r)-depth(q) & \text{where } r \text{ is in } set_1(q),\\ k_{\min}. \end{cases}$$

The function $d_1[q]$ is the smallest difference in depth between q and any state r for which $path(r)$ is the string $u\,path(q)$ for some nonnull u. ($d_1[q]$ is in fact the length of u.) In other words, if some reversed keyword contains another reoccurrence of substring $path(q)$ at the point at which a mismatch occurs, it is safe to move the start state to the right by $d_1[q]$ positions.

The second shift function is defined to align a suffix of $s_{i-j+1}s_{i-j+2}\dots s_i$ (which is $path(q)$ reversed) with a prefix of some keyword:

$$d_2[q_0]=k_{\min},$$

$$d_2[q]=\min\begin{cases} depth(r)-depth(q) & \text{where } r \text{ is in } set_2(q),\\ d_2[parent(q)]. \end{cases}$$

The value of $d_2[q]$ is determined by those keywords w such that $path(q)$, or a prefix thereof, is a suffix of w^{R}, the reversal of w. Table 2 summarizes this information for the trie above.

Matching proceeds somewhat like in the Boyer–Moore algorithm with the trie playing the role of the single keyword. Define

$$char(c)=\min\begin{cases} depth(q) & \text{where the transition into } q \text{ is labeled } c,\\ k_{\min}+1. \end{cases}$$

Table 2
Shift functions for the trie in Fig. 13

node	d_1	d_2	set_1	set_2
0	1	3		
1	1	2	{2, 5, 8, 10, 13, 16}	{8, 13, 16}
2	3	2	∅	∅
3	3	2	∅	∅
4	3	2	∅	∅
5	3	2	∅	∅
6	3	2	∅	∅
7	1	2	{3, 11}	∅
8	3	2	∅	∅
9	1	3	{3, 7, 11}	∅
10	1	1	{8}	{8}
11	3	1	∅	∅
12	3	1	∅	∅
13	3	1	∅	∅
14	3	1	∅	∅
15	2	3	{4, 12}	∅
16	2	2	{5, 13}	{13}

Finally, let

$$shift[q, s_{i-j}, j]=\min\{\max(d_1[q], char(s_{i-j})-j-1), d_2[q]\}.$$

This function is used if there is a mismatch in state q at input character s_{i-j}. The $char(s_{i-j})-j-1$ term sometimes allows us to shift more than $d_1[q]$ positions by observing that d_1 is at most $k_{\min}$ and that the shortest distance from the root to a node whose incoming edge is labeled by s_{i-j} may be greater than this. In this case, it is safe to advance by $char(s_{i-j})-j-1$ positions (the term $-j-1$ adjusts for the relative position of the start state).

Note that we use the minimum of d_2 and the maximum of d_1 and $char$ in computing $shift$. At node 14 in the trie of Fig. 13, for example, $d_1[14]=3=k_{\min}$ because $set_1(14)$ is empty. When we reach node 14, $s_{i-j+1}s_{i-j+2}\dots s_i = ccbab$ with the start state over the rightmost b. It might appear that we could shift the start state 3 positions to the right once we reach node 14. But $d_2[14]=1$ because of node 10 where we shift by 1 to check for a match of aba; thus, at any descendant of node 10 we cannot shift by more than 1 in case we miss this possible match.

In general, a shift to the right by $shift[q, s_{i-j}, j]$ positions is guaranteed not go past an occurrence of a keyword. The entire algorithm is summarized in Fig. 14.

```
begin
  $q := q_0$
  $i := k_{\min}$
  $j := 0$
  while $i \leqslant n$ do
  begin
    while $j < i$ cand $g[q, s_{i-j}] \neq fail$ do
    begin
      $q := g[q, s_{i-j}]$
      $j := j+1$
      if $q$ is accepting then return "yes"
    end
    $i := i + shift[q, s_{i-j}, j]$
    $q := q_0$
    $j := 0$
  end
  return "no"
end
```

Fig. 14. The Commentz-Walter algorithm for matching multiple keywords.

The Commentz-Walter pattern-matching automaton can be constructed in $O(m)$ time using techniques similar to those in Section 4.1, but its worst-case running time on an input string of length n is $\Theta(mn)$. In practice, with small numbers of keywords, the Boyer–Moore aspect of the Commentz-Walter algorithm can make it faster than the Aho–Corasick algorithm, but with larger numbers of keywords the Aho–Corasick algorithm has a slight edge. It is possible at a greater than linear-time preprocessing cost to produce a version of the Commentz-Walter algorithm whose worst-case behavior is linear in n [37].

5. Matching regular expressions

We now generalize the pattern-matching problem to full regular expressions.

5.1. PROBLEM. Given a pattern consisting of a regular expression r and an input string $s=s_1s_2 \ldots s_n$ answer "yes" if r matches a substring of s; "no" otherwise. We assume the length of r is m.

This section describes two algorithms to solve this problem. In the first the pattern-matcher generator constructs a nondeterministic finite automaton, which is then used as the matcher to process the input string. In the second the generator constructs a deterministic finite automaton. Both approaches have been used in regular-expression pattern-matching programs.

5.1. Nondeterministic recognizers for regular expressions

This section outlines a regular-expression pattern-matching technique originally due to Thompson [127] that was used in the text editor *qed* and in a simplified form in the UNIX system command *grep*. We will prepend to the given regular expression the expression $(a_1|a_2|\ldots|a_f)^*$ where a_1 through a_f are the symbols of the input alphabet. This prefix allows matching to begin at any position in the input string. For the remainder of this section, we will assume r contains this prefix.

We begin by constructing a nondeterministic finite automaton (NDFA) from r. An NDFA is a directed graph in which the nodes are the states and each edge is labeled by a single character or the symbol ε, which stands for the empty string. One state is designated as an *initial* state, and some states as *accepting* states. An NDFA accepts (matches) a string if there is a path from the start state to an accepting state whose edge labels spell out the string.

Once we have constructed an NDFA for r, we run it on the input string s. If the NDFA enters an accepting state while processing s, we report that r matches s, otherwise, we report "no". The NDFA can take the form of an executable program or a state-transition table that is interpreted.

The recursive procedure below can be used to construct an NDFA for the regular expression. We first parse the regular expression into its constituent subexpressions according to the formation rules used to define a regular expression in Section 2.1. Using rule (1), we construct an NDFA for a non-metacharacter. Rules (2)–(5) show how to combine the NDFAs constructed from the constituent subexpressions.

(1) For a non-metacharacter c, construct the NDFA in Fig. 15(a) where i is a new initial state and a a new accepting state. This automaton clearly accepts exactly the string c.

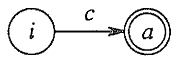

Fig. 15(a).

(2) Suppose N_{r_1} and N_{r_2} are NDFAs for r_1 and r_2. For the regular expression $r=r_1|r_2$, construct the NDFA N in Fig. 15(b) where i is a new initial state and a a new accepting state. There is an ε-transition from i to the start states of N_{r_1} and N_{r_2}. There is an ε-transition from the accepting states of N_{r_1} and N_{r_2} to the new accepting state a. (The initial and accepting states of N_{r_1} and N_{r_2} are not considered start or accepting states of N_r.) Note that any path from i to a must pass through either N_{r_1} or N_{r_2}. Thus, N accepts any string accepted by N_{r_1} or N_{r_2}.

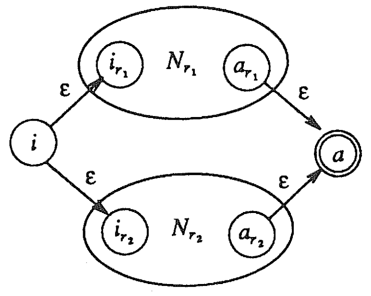

Fig. 15(b).

(3) Suppose N_{r_1} and N_{r_2} are NDFAs for r_1 and r_2. For the regular expression $r=r_1r_2$, construct the NDFA N in Fig. 15(c) where the start state of N_{r_1} becomes the start state of N and the accepting state of N_{r_2} becomes the accepting state of N. The accepting state of N_{r_1} is merged with the start state of N_{r_2}; that is, all transitions from the start state of N_{r_2} become transitions from the accepting state of N_{r_1}. The new merged state loses its status as a start or accepting state in N. A path from i_{r_1} to a_{r_2} must go first through N_{r_1} and then through N_{r_2}, so N accepts any string of the form xy where N_{r_1} accepts x and N_{r_2} accepts y.

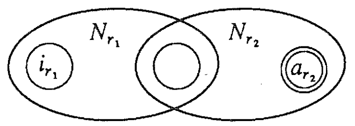

Fig. 15(c).

(4) Suppose N_{r_1} is an NDFA for r_1. For the regular expression $r=r_1^*$, construct the NDFA N in Fig. 15(d) where i is a new initial state and a a new accepting state. In N, we can go from i to a directly, along an edge labeled ε, representing the fact that s^* matches the empty string, or we can go from i to a passing through N_{r_1} one or more times. Thus, N accepts any string matched by r_1^*.

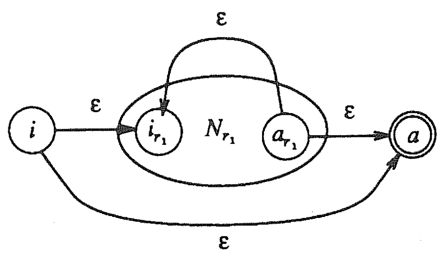

Fig. 15(d).

(5) For the regular expression (r) use the NDFA for r.

Figure 16 shows the NDFA that results from this construction for the regular expression $(a|b)^*aba$. This construction produces an NDFA N for r with the following properties:

(1) N has at most twice as many states as the length of r, since each step of the construction creates at most two new states.

(2) N has exactly one start state and one accepting state, and the accepting state has no outgoing transitions. This property holds for each of the constituent NDFAs as well.

(3) Each state has either one outgoing edge labeled by a character or at most two outgoing ε-edges.

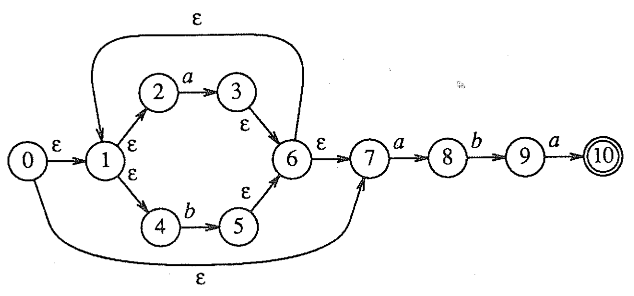

Fig. 16. NDFA for $(a|b)^*aba$.

Figure 17 contains an algorithm to determine whether an NDFA N with initial state i and accepting state a matches a substring of an input string $s=s_1s_2\ldots s_n$. The algorithm uses the function *epsilon*(Q) to compute all states that can be reached from a set of states Q by following only ε-edges. After reading each character, the algorithm determines Q, the current set of states for N. It computes the next set of states from Q in two stages. First, it determines *goto*(Q, c), all states that can be reached from a state in Q by a transition on c, the current input character. Then, it computes *epsilon*(*goto*(Q, c)), all states that can be reached from *goto*(Q, c) by following only ε-edges. The algorithm returns "yes" if the accepting state is in the set of current states. If no accepting state is ever encountered, "no" is returned.

```
begin
  Q := epsilon({i})
  if Q contains a then return "yes"
  for j := 1 to n do
  begin
    Q := epsilon(goto(Q, s_j))
    if Q contains a then return "yes"
  end
  return "no"
end
```

Fig. 17. Algorithm to simulate an NDFA on an input string $s_1 \ldots s_n$.

The NDFA can be simulated in time proportional to $|N| \times |s|$, where $|N|$ is the number of states in N and $|s|$ is the length of s, by taking advantage of the special properties of N. The set of states reachable after each input character can be efficiently computed using two stacks and a bit vector indexed by states. One stack is used to store Q, the current set of states, and the other stack to determine the next set of states. Since each state has at most two out-transitions, each state on the first stack can add at most two new states to the second stack. The bit vector is used to quickly determine whether a state is already on the second stack so that we do not add it twice. Once we have put the states in *goto*(Q, c) onto the second stack, we can use a simple reachability algorithm to compute *epsilon*(*goto*(Q, c)). When we have computed all the reachable states on the second stack, we read the next input character and interchange the roles of the two stacks.

Since there can be at most $|N|$ states on a stack, the computation of the next set of states from the current set of states can be done in time proportional to $|N|$. Thus, the time needed to run N on input s is proportional to $|N| \times |s|$. Since the number of states in N is at most twice the length of r, the running time of this algorithm is $O(|r| \times |s|)$. Thus, we have the following theorem.

5.2. THEOREM. *Problem* 5.1 *can be solved with an NDFA in* $O(mn)$ *time and* $O(m)$ *space.*

Myers [105] has shown how the use of node listings and the "Four Russians" trick [6] can be applied to this algorithm to derive an $O(mn/\log n)$ time and space solution to Problem 5.1.

5.2. *Deterministic recognizers for regular expressions*

Although the algorithm in the previous section produces a compact NDFA, its running time is proportional to the product of the size of the regular expression and the length of the input. A deterministic finite automaton (DFA) is an NDFA in which there are no ε-transitions and in which every state has at most one transition on any input character. DFAs are well suited for regular-expression pattern matching because they are capable of recognizing all regular-expression patterns. Moreover, a DFA can be simulated in real time because an input character causes at most one state transition.

This section describes a DFA-based regular-expression pattern-matching algorithm. The algorithm first constructs a syntax tree for the regular expression, such as the one in Fig. 18 for the regular expression $(a|b)^*aba\#$. The symbol $\#$ is an endmarker appended to the expression; its function will be explained shortly. A dot is used to represent the concatenation operator.

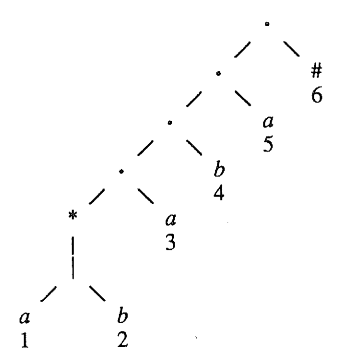

Fig. 18. Syntax tree for $(a|b)^*aba\#$.

The leaves of the syntax tree are labeled by the non-metacharacters in the regular expression. Using a technique suggested by McNaughton and Yamada [97] we associate with each leaf a unique integer called the *position* of the leaf. Positions are shown below the leaves in the syntax tree of Fig. 18.

The transitions of the DFA are constructed directly from the syntax tree. Each state of the DFA is the set of positions corresponding to the leaves that are active after having read some prefix of the input string. Initially, leaves 1, 2, and 3 are active so the initial state of the DFA is the set $\{1, 2, 3\}$.

The next state Q' representing the transition from state Q on an input character c is computed as follows. For each position i in Q whose leaf-symbol matches c, we add $follow(i)$ to T, where $follow(i)$ is the set of positions that can "follow" the leaf labeled by i in the syntax tree. These two rules define $follow(i)$:

(1) If *left* and *right* are the two children of a node labeled by a concatenation operator in the syntax tree and i is a position that can last be active in the subtree rooted at *left*, then all positions initially active in the subtree rooted at *right* are in $follow(i)$.

(2) If i is a position that can last be active in a subtree rooted at a $*$-node in the syntax tree, then all positions initially active in that subtree are in $follow(i)$.

Let us compute the state Q representing the transition from the initial state $\{1, 2, 3\}$ on the input symbol a. The leaf corresponding to position 1 matches a, so we add $follow(1)=\{1, 2, 3\}$ to Q. Position 2 does not match a, but position 3 does, so we add $follow(3)=\{4\}$ to Q. Thus the transition from state $\{1, 2, 3\}$ on a is to state $\{1, 2, 3, 4\}$.

To create the complete DFA, we compute all transitions for each new state. Any state containing the position corresponding to the endmarker $\#$ is made an accepting state.

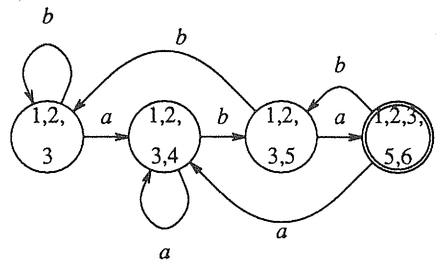

Fig. 19. DFA for $(a|b)^*aba$.

We do not compute transitions for the symbol $\#$. Figure 19 contains the complete DFA for the regular expression $(a|b)^*aba$.

The transition function of a DFA can be simply represented as a two-dimensional array. A DFA can then be simulated efficiently by the table look-up program in Fig. 20.

```
begin
  Q := InitialState
  if Q is accepting then return "yes"
  for j := 1 to n do
  begin
    Q := goto[Q, s_j]
    if Q is accepting then return "yes"
  end
  return "no"
end
```

Fig. 20. Algorithm to simulate a DFA on an input string $s_1 \ldots s_n$.

Thus, once a DFA has been constructed from a regular expression, we can determine whether it matches the input string in $O(n)$ time. Note that neither the size of the input alphabet nor the length of the regular expression affect the time to simulate the DFA. Berry and Sethi [26] relate this technique to the derivatives method for constructing a DFA from a regular expression proposed by Brzozowski [33].

If the transition function is stored as a two-dimensional array, then the storage requirement for a DFA is the product of the input alphabet size times the number of states. For some regular expressions even a minimum-state DFA recognizer may have 2^n states, where n is the length of the regular expression. In these situations, the complete DFA is time-consuming to construct, and the transition function requires lots of storage. Consider, for example, the regular expression

$$(a|b)^*a(a|b)(a|b)\ldots(a|b)$$

where there are k copies of $(a|b)$ at the end. This expression denotes all sequences of a's and b's in which the $(k+1)$st symbol from the end is an a. The smallest DFA recognizing this expression must have at least 2^k states to remember the 2^k possible sequences of the last k characters it has seen. Thus, to summarize, we have the following theorem.

5.3. THEOREM. *Problem* 5.1 *can be solved with a DFA in* $O(2^m+n)$ *time and* $O(2^m)$ *space.*

Several techniques are available to reduce the space requirements of the transition table [9]. An effective storage-reduction technique that has been used by the author in the UNIX system command *egrep* is "lazy transition evaluation." The transition function is only computed when the DFA is run. Computed transitions are kept in a cache. Before a transition is made, the cache is examined. If the required transition is not in the cache, it is computed and stored for subsequent use. If the cache becomes full, some or all of the previously computed transitions are removed to make room for the new transition. This method allows the pattern matcher to use a fixed-size storage area for the transitions with only a small run-time penalty. The observed performance in practice of this approach for solving Problem 5.1 is $O(m+n)$ time and $O(m)$ space, combining the best features of both the nondeterministic and deterministic approaches. It would be interesting to know whether it is possible to construct a regular-expression pattern-matching algorithm with this as its worst-case behavior.

The problem of storing a transition table compactly and yet providing constant-time access to its elements has stimulated recent research in perfect hashing [51, 126] and dynamic perfect hashing [8, 45]. Some additional issues in the efficient representation of transition tables are discussed in [12]. An alternative to creating a transition table is to generate machine code directly to simulate the automaton [109, 127].

An *extended* regular expression is one that also has operators for intersection and complement. Hopcroft and Ullman describe a dynamic programming algorithm to match extended regular expressions in time $O((n+m)^4)$ [70, Exercise 3.23].

6. Related problems

In the three previous sections we presented pattern-matching algorithms for keywords, sets of keywords, and regular expressions. There are many other string-matching problems that are of interest in computer science and in this section we will mention a few that are related to the ones that we have studied.

6.1. *Matching regular expressions with back referencing*

Let us briefly consider the problem of matching regular expressions with back referencing because it dramatically brings out the point that as we generalize the class of patterns, we can make the pattern-matching process much more difficult computationally.

6.1. PROBLEM. Given a pattern consisting of a rewbr r and an input string s, answer "yes" if s contains a substring matched by r; "no" otherwise.

6.2. THEOREM. *Problem* 6.1 *is* NP-*complete.*

PROOF. We will use a reduction from the vertex-cover problem. Let $E_1, E_2, \ldots, E_m$ be subsets of cardinality 2 of some finite set of vertices V. The vertex-cover problem is to determine, given a positive integer k, whether there exists a subset V' of V of cardinality at most k such that V' contains at least one element in each E_i. We can think of the E_i's as being edges of a graph and V' as being a set of vertices such that each edge contains at least one vertex in V'. The vertex-cover problem is a well-known NP-complete problem [59].

We can transform this problem into a pattern-matching problem for rewbrs as follows. Let N be the parenthesized string $(n_1|n_2|\ldots|n_f)$ where $V=\{n_1, n_2, \ldots, n_f\}$. Let $\#$ be a distinct marker symbol. For $1 \leqslant i \leqslant k$, let

$$x_i = N^*N\%v_i\,N^*\#$$

where the v_i's are distinct variable names. Likewise, for $1 \leqslant i \leqslant m$, let

$$y_i = N^*N\%w_i\,N^*\#$$

where the w_i's are distinct variable names. For $1 \leqslant i \leqslant m$, let

$$z_i = w_i^* v_1^* v_2^* \ldots v_k^* w_i^* \#$$

Let r be the rewbr $x_1 \ldots x_k y_1 \ldots y_m z_1 \ldots z_m$.

We shall now construct an input string s such that if the rewbr r matches s, then the vertex-cover problem has a solution of size k. Let u be the string $n_1 n_2 \ldots n_f\,\#$ repeated $k+m$ times. Let e_i be the string $ab\#$ where $E_i = \{a, b\}$ for $1 \leqslant i \leqslant m$. Finally, let s be the input string $ue_1 \ldots e_m$.

Now, notice that r matches s if and only if the set of vertices assigned to the variables $v_1, \ldots, v_k$ forms a vertex cover for the set of edges $\{E_1, E_2, \ldots, E_m\}$. Thus, r matches s if and only if the vertex-cover problem has a solution of cardinality at most k.

It is easy to match a rewbr in nondeterministic polynomial time. The most straightforward approach to matching a rewbr pattern r deterministically is to use backtracking to keep track of the possible substrings of the input string s that can be assigned to the variables in r. There are $O(n^2)$ possible substrings that can be assigned to any one variable in r, where n is the length of s. If there are k variables in r, then there are $O(n^{2k})$ possible assignments in all. Once an assignment of substrings to variables is fixed, the problem reduces to ordinary regular expression matching. Thus, rewbr matching can be done in at worst $O(n^{2k})$ time. $\square$

This NP-completeness result implies that if $P \neq NP$, then there is no polynomial-time algorithm for matching rewbrs.

6.2. *Finding repeated patterns and palindromes*

Rewbrs allow one to find repeated patterns in an input string, but at a considerable cost. There are much more efficient techniques for finding certain classes of repeated patterns. Weiner [137] devised an efficient way to construct a compact index to all the distinct substrings of a set of strings in linear time. Variants of this index are called *position trees* [6, 91], *subword trees* [13], *complete inverted files* [28–31], *PATRICIA trees* [103], and *suffix trees* [41, 95, 115]. The position tree is useful for solving in linear time problems such as finding the longest repeated substring of an input string. Apostolico [13] discusses other innovative uses for this index. Chen and Seiferas [34] have a particularly clean construction for the index. Crochemore [42] and Perrin [110] relate the failure function of Section 5.1 to the construction of a minimal suffix automaton.

A related problem, more of mathematical interest, is to find all the squares in a string, where a *square* is a substring of the form xx with x nonempty. For example, the Fibonacci string F_n contains at least $(|F_n| \log |F_n|)/12$ different squares. Shortly after the turn of the century, Thue [128, 129] had asked how long a square-free string could be and showed that with an alphabet of three characters square-free strings of any length could be constructed. Several authors have devised $\Theta(n \log n)$ algorithms for finding all squares in a string of length n [20, 40, 89, 90]. One can determine in linear time whether a string s has a prefix that is a square, that is, whether $s = xxy$ for nonempty x and some y. It is an open problem as to whether the set of strings of the form xxy (*prefixsquares*) can be recognized by a 2DPDA. Rabin [112] gives a simple "fingerprinting" algorithm similar to the one in Section 3.2 for finding in $O(n \log n)$ expected time the earliest repetition in a string s, that is, the shortest w and x such that $s = wxxy$.

Palindromes, strings that read the same forwards and backwards, have provided amusement for centuries. Strings that begin with an even-length palindrome, that is, strings of the form $xx^{R}y$ with x nonempty, can be recognized by a 2DPDA, and hence in linear time on a RAM. (x^{R} is the reversal of x.) Strings of nontrivial palindromes, that is, strings of the form $x_1 x_2 \ldots x_n$ (*palstars*), where each x_i is a palindrome of length greater than 1, can be recognized in linear time on a RAM but we do not know whether they can be recognized by a 2DPDA. See [52, 54, 56–58, 92, 121, 123] for more details and for algorithms that give real-time performance for finding palindromes in strings.

6.3. *Approximate string matching*

We now consider several important variants of pattern-matching problems that arise in areas such as file comparison, molecular biology, and speech recognition. Perhaps the simplest is the *file-difference* problem: Given two strings x and y, determine how "close" x is to y. A useful way to compare the two files is to print a minimal sequence of editing changes that will convert the first file into the second. Let us treat the files as two strings of symbols, $x = a_1 \ldots a_m$ and $y = b_1 \ldots b_n$, where a_i represents the ith line of the first file and b_j represents the jth line of the second. Define two editing transformations: the insertion of a line and the deletion of a line. The *edit distance* between the two files is the smallest number of editing transformations

required to change the first file into the second. For example, if the first file is represented by the string *abcabba* and the second by *cbabac*, the edit distance is 5.

A simple dynamic programming algorithm to compute this edit distance has been discovered independently by many authors [118]. Let d_{ij} be the edit distance between the prefix of length i of the first string and the prefix of length j of the second string. Let d_{00} be 0, d_{i0} be i for $1 \leqslant m$, and d_{0j} be j for $1 \leqslant j \leqslant n$. Then, for $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$, compute d_{ij} by taking the minimum of the three quantities:

$$(1)\ d_{i-1,j}+1, \qquad (2)\ d_{i,j-1}+1, \qquad (3)\ d_{i-1,j-1} \text{ if } a_i = b_j$$

The first quantity represents the deletion of the jth character from the first string, the second represents the insertion of a character after the $(j-1)$st position in the first string, and the third says $d_{ij} \leqslant d_{i-1,j-1}$ if the ith character in the first string agrees with the jth character in the second string. After completing this computation, we can easily show that d_{mn} gives the minimum number of edit changes required to transform the first file into the second. From these distance calculations we can construct the corresponding sequence of editing transformations. Figure 21 shows the matrix of edit distances for the strings *abcabba* and *cbabac*.

$$\begin{array}{cc|cccccccc}
\mathtt{c} & 6 & 6 & 5 & 4 & 3 & 4 & 5 & 6 & 5 \\
\mathtt{a} & 5 & 5 & 4 & 3 & 4 & 3 & 4 & 5 & 4 \\
\mathtt{b} & 4 & 4 & 3 & 2 & 3 & 4 & 3 & 4 & 5 \\
\mathtt{a} & 3 & 3 & 2 & 3 & 4 & 3 & 4 & 5 & 4 \\
\mathtt{b} & 2 & 2 & 3 & 2 & 3 & 4 & 3 & 4 & 5 \\
\mathtt{c} & 1 & 1 & 2 & 3 & 2 & 3 & 4 & 5 & 6 \\
 & 0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
 & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 & & & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a}
\end{array}$$

Fig. 21. Edit distances d_{ij}.

The running time of this algorithm is always proportional to mn, the product of the sizes of the two files. In one sense, this algorithm minimizes the number of character comparisons: with equal-unequal comparisons, $\Omega(mn)$ character comparisons are necessary in the worst case [5].

Hunt and McIlroy [74] implemented Hirschberg's linear-space version of this algorithm [66] which worked very well for short inputs, but when it was applied to longer and longer files its performance became significantly slower. This behavior, of course, is entirely consistent with the quadratic nature of this algorithm's time complexity.

Shortly thereafter, Hunt and Szymanski [75] proposed another approach to the file-difference problem. They suggested extracting a longest common subsequence from the two strings and producing the editing changes from the subsequence.

DEFINITION. A *subsequence* of a string x is a sequence of characters obtained by deleting zero or more characters from x. A *common subsequence* of two strings x and y is a string that is a subsequence of both x and y, and a *longest common subsequence* is a common subsequence that is of greatest length. For example, *baba* and *cbba* are both longest common subsequences of *abcabba* and *cbabac*.

The problem of finding a longest common subsequence of two strings is closely related to the file-difference problem. The following formula shows the relationship between the edit distance and the length of a longest common subsequence of two strings of symbols, $x=a_1 \dots a_m$ and $y=b_1 \dots b_n$:

$$EditDistance(x, y) = m+n-2length(lcs(x, y)).$$

To solve the longest-common-subsequence problem for x and y, Hunt and Szymanski constructed a matrix M, where M_{ij} is 1 if $a_i=b_j$ and 0 otherwise. They produced a longest common subsequence for x and y by drawing a longest strictly monotonically increasing line through the points of the M matrix as shown in Fig. 22. If there are r points, such a line can be found in time proportional to $(r+n)\log n$, assuming $n \geqslant m$. In the file-difference problem, r is usually on the order of n, so in practice the running time of the Hunt–Szymanski technique is $O(n \log n)$, significantly faster than the dynamic programming solution.

McIlroy put this new algorithm into his program and the inputs that used to take

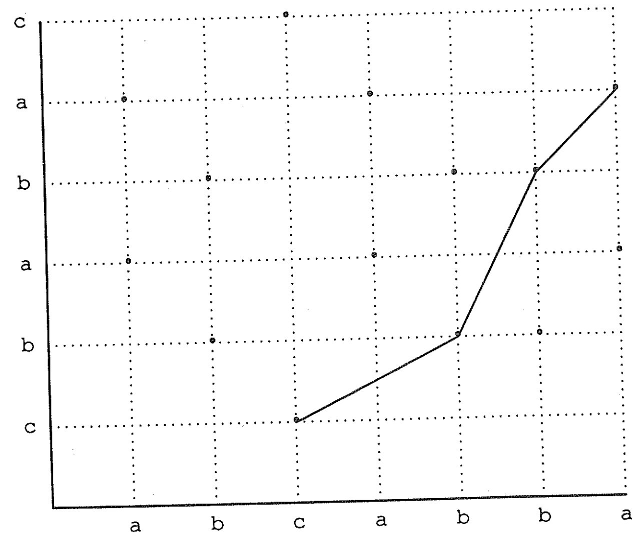

Fig. 22. A longest monotonically increasing line represents a longest common subsequence.

many minutes to compare now ran in a few seconds. The functionality of the program had not changed but its average performance had improved noticeably.

More recently, Myers [104] and Ukkonen [131] independently suggested a third algorithm for this problem: treat the problem as one of constructing a cheapest-cost path in a graph. The nodes of the graph consist of the intersection points on an $(m+1)\times(n+1)$ grid. In addition to the horizontal and vertical edges, there is a diagonal edge from node $(i-1, j-1)$ to (i, j) if $a_i=b_j$. Each horizontal and vertical edge has a cost of 1, and each diagonal edge a cost of 0. A cheapest path from the origin to node (m, n) uses as many diagonal edges as possible as illustrated in Fig. 23. The tails of the diagonal edges on the cheapest path define a longest common subsequence between the two strings. Dijkstra's algorithm [6] can be used to construct the cheapest path in time proportional to dn, where d is the edit distance between the two strings. In situations where d is small, this approach is the method of choice.

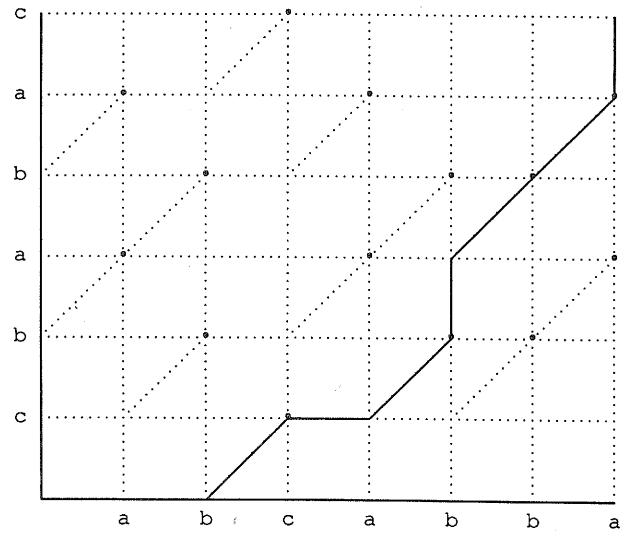

Fig. 23. A cheapest path represents a longest common subsequence.

Myers [103] describes a refinement of the algorithm above that runs in $O(n \log n + d^2)$ time using suffix trees. Masek and Paterson [93] present an $O(n^2/\log n)$ time algorithm using the "Four Russians" trick. So far, no single algorithm is known for the longest-common-subsequence problem that dominates all applications. More algorithms, programs, and other aspects of the problem are discussed in [10, 14, 15, 19, 36, 50, 64, 66, 67, 68, 101, 107, 130, 138].

Many generalizations of the file-difference problem arise in molecular biology, speech recognition, and related applications. We can define the *approximate string-matching* problem as follows: Given a pattern p, an input string s, an integer k, and a distance metric d, find all substrings x of s such that $d(p, x) \leqslant k$, that is, the distance between the pattern p and the substring x is at most k.

When p is a single string, several distance measures have been proposed. The two most common are the Hamming distance and Levenshtein distance. The *Hamming distance* between two strings of equal length is the number of positions with mismatching characters. For example, the Hamming distance between *abcabb* and *cbacba* is 4. The approximate string-matching problem with d being Hamming distance is called *string matching with k mismatches*. Algorithms for the k-mismatches problem are presented in [55, 76, 82].

The *Levenshtein distance* between two strings of not necessarily equal length is the minimum number of character changes, insertions, and deletions required to transform one string into the other. For example, the Levenshtein distance between *abcabba* and *cbabac* is 4. Algorithms for computing the Levenshtein distance between a pair of strings are contained in [91, 118, 131, 134]. The approximate string-matching problem with d being the Levenshtein distance is called *string matching with k differences*. Algorithms for the k-differences problem are described in [81, 83, 118, 132, 135].

Galil and Giancarlo [55] present an excellent survey of algorithms and data structures that have been devised to solve approximate string-matching problems efficiently. More general distance measures are considered in [1, 47, 87, 100]. The approximate string-matching problem where p is a regular expression has been considered by Wagner [133], Wagner and Seiferas [135], and Myers and Miller [106].

6.4. *String matching with "don't cares"*

We conclude by mentioning a variant of the string-matching problem in which there is a "don't care" symbol that matches any single character (including a don't care symbol). We say two strings $a_1 \ldots a_m$ and $b_1 \ldots b_m$ match if for $1 \leqslant i \leqslant m$ whenever a_i and b_i are not both don't cares, they are equal. The pattern-matching problem with don't cares comes in two flavors:

(1) only the pattern p contains don't cares, and

(2) both the pattern p and the input string s contain don't cares.

Pinter [111] uses an extension of the Aho–Corasick algorithm to solve the first problem. Fischer and Paterson [49] have shown that the second problem reduces to integer multiplication. If $M(m, n)$ is the amount of time needed to multiply an m-bit number by an n-bit number, then the time needed to solve the second version of the pattern-matching problem with don't cares is

$$O(M(|p|, |s|) \log|p| \log A)$$

where A is the size of the input alphabet. Using the Schönhage–Strassen integer-multiplication algorithm [6], the time bound becomes

$$O(|s| \log^2|p| \log\log|p| \log A).$$

7. Concluding remarks

In this chapter we have discussed algorithms for solving string-matching problems that have proven useful for text-editing and text-processing applications. As the reader can see, even in this restricted domain there is a rich literature with interesting ideas and deep analyses.

We should mention a few important generalizations of string pattern matching that we have not had the space to consider. There is an abundant literature on term-rewriting systems [72], and on efficient parsing methods for context-free grammars [70] and special cases of context-free grammars, such as the LL(k) and LR(k) grammars [6]. There are also methods for matching patterns in trees, graphs, and higher-dimensional structures [6, 27, 69, 77, 117].

In this paper, we have restricted ourselves to algorithms for the RAM of computation. In recent years, there has been accelerating interest in developing algorithms for various parallel models of computation, but practical experience with these algorithms has been much more limited than with those for the RAM. In the coming years, we hope to see for these new models of computation the same interplay between theory and practice that has enriched the field of pattern-matching algorithms for the random-access machine.

Acknowledgment

The author is grateful to David Lee for helpful comments on the manuscript.

References

[1] ABRAHAMSON, K., Generalized string matching, *SIAM J. Comput.* **16**(6) (1987) 1039–1051.

[2] AHO, A.V., Pattern matching in strings, in: R. Book, ed., *Formal Language Theory, Perspectives and Open Problems* (Academic Press, New York, 1980) 325–347.

[3] AHO, A.V., and M.J. CORASICK, Efficient string matching: an aid to bibliographic search, *Comm. ACM* **18**(6) (1975) 333–340.

[4] AHO, A.V., M. GANAPATHI and S.W.K. TJIANG, Code generation using tree matching and dynamic programming, *ACM Trans. Programming Languages and Systems* **11**(4) (1989).

[5] AHO, A.V., D.S. HIRSCHBERG and J.D. ULLMAN, Bounds on the complexity of the maximal common subsequence problem, *J. ACM* **23**(1) (1976) 1–12.

[6] AHO, A.V., J.E. HOPCROFT and J.D. ULLMAN, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[7] AHO, A.V., B.W. KERNIGHAN and P.J. WEINBERGER, *The AWK Programming Language* (Addison-Wesley, Reading, MA, 1988).

[8] AHO, A.V. and D. LEE, Storing a dynamic sparse table, in: *Proc. 27th IEEE Symp. on Foundations of Computer Science* (1986) 56–60.

[9] AHO, A.V., R. SETHI, and J.D. ULLMAN, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA, 1986).

[10] ALLISON, L. and T.I. DIX, A bit-string longest-common-subsequence algorithm, *Inform. Process. Lett.* **23** (1986) 305–310.

[11] ANGLUIN, D., Finding patterns common to a set of strings, in: *Proc. 11th Ann. ACM Symp. on Theory of Computing* (1979) 130–141.
[12] AOE, J., Y. YAMAMOTO and R. SHIMADA, A method for improving string pattern matching machines, *IEEE Trans. Software Engrg.* **10**(1) (1984) 116–120.
[13] APOSTOLICO, A., The myriad virtues of subword trees, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985).
[14] APOSTOLICO, A., Improving the worst-case performance of the Hunt–Szymanski strategy for the longest common subsequence of two strings, *Inform. Process. Lett.* **23** (1986) 63–69.
[15] APOSTOLICO, A., Remark on the Hsu–Du new algorithm for the longest common subsequence problem, *Inform. Process. Lett.* **25** (1987) 235–236.
[16] APOSTOLICO, A. and Z. GALIL, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985).
[17] APOSTOLICO, A. and R. GIANCARLO, Pattern matching machine implementation of a fast test for unique decipherability, *Inform. Process. Lett.* **18** (1984) 155–158.
[18] APOSTOLICO, A. and R. GIANCARLO, The Boyer–Moore–Galil string searching strategies revisited, *SIAM J. Comput.* **15**(1) (1986) 98–105.
[19] APOSTOLICO, A. and G. GUERRA, The longest common subsequence problem revisited, *Algorithmica* **2** (1987) 315–336.
[20] APOSTOLICO, A. and F.P. PREPARATA, Optimal off-line detection of repetitions in a string, *Theoret. Comput. Sci.* **22** (1983) 297–315.
[21] BAEZA-YATES, R.A., Improved string searching, *Software—Practice and Experience* **19**(3) (1984) 257–271.
[22] BARTH, G., An alternative for the implementation of Knuth–Morris–Pratt algorithm, *Inform. Process. Lett.* **13** (1981) 134–137.
[23] BARTH, G., Relating the average-case costs of the brute-force and the Knuth–Morris–Pratt string matching algorithm, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985).
[24] BEAN, D.R., A. EHRENFEUCHT and G.F. MCNULTY, Avoidable patterns in strings of symbols, *Pacific J. Math.* **85** (1985) 31–56.
[25] BEN-YEHUDA, S. and R.Y. PINTER, Symbolic layout improvement using string matching based local transformations, in: *Proc. Decennial Caltech Conf. on VLSI* (MIT Press, Cambridge, MA, 1989) 227–239.
[26] BERRY, G. and R. SETHI, From regular expressions to deterministic automata, *Theoret. Comput. Sci.* **48**(1) (1986) 117–126.
[27] BIRD, R.S., Two dimensional pattern matching, *Inform. Process. Lett.* **6**(5) (1977) 168–170.
[28] BLUMER, A., J. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, M.T. CHEN and J. SEIFERAS, The smallest automaton recognizing the subwords of a text, *Theoret. Comput. Sci.* **40**(1) (1985) 31–56.
[29] BLUMER, A., J. BLUMER, A. EHRENFEUCHT, D. HAUSSLER and R. MCCONNELL, Building a complete inverted file for a set of text files in linear time, in: *Proc. 16th ACM Symp. on Theory of Computing* (1984) 349–358.
[30] BLUMER, A., J. BLUMER, A. EHRENFEUCHT, D. HAUSSLER and R. MCCONNELL, Building the minimal DFA for the set of all subwords of a word on-line in linear time in: J. Paredaens, ed., *Proc. 11th Internat. Coll. on Automata, Languages and Programming*, Lecture Notes in Computer Science, Vol. 172 (Springer, Berlin, 1984) 109–118.
[31] BLUMER, A., J. BLUMER, D. HAUSSLER, R. MCCONNELL and A. EHRENFEUCHT, Complete inverted files for efficient text retrieval and analysis, *J. ACM* **34**(3) (1987) 578–595.
[32] BOYER, R.S. and J.S. MOORE, A fast string searching algorithm, *Comm. ACM* **20**(10) (1977) 62–72.
[33] BRZOZOWSKI, J.A., Derivatives of regular expressions, *J. ACM* **11**(4) (1964) 481–494.
[34] CHEN, M.T. and J.I. SEIFERAS, Efficient and elegant subword tree construction, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985).
[35] CHERRY, L.L., Writing tools, *IEEE Trans. Comm.* **30**(1) (1982) 100–105.
[36] CHVATAL, V. and D. SANKOFF, Longest common subsequence of two random sequences, *J. Appl. Probab.* **12** (1975) 306–315.
[37] COMMENTZ-WALTER, B., A string matching algorithm fast on the average, in: H.A. Maurer, ed., *Proc. 6th Internat. Coll. on Automata, Languages and Programming* (Springer, Berlin, 1979) 118–132.

[38] COOK, S.A., Linear time simulation of deterministic two-way pushdown automata, in: *Proc. IFIP Congress, 71, TA-2* (North-Holland, Amsterdam, 1971) 172–179.
[39] CHROBAK, M. and W. RYTTER, Remarks on string-matching and one-way multihead automata, *Inform. Process. Lett.* **24** (1987) 325–329.
[40] CROCHEMORE, M., An optimal algorithm for computing the repetitions in a word, *Inform. Process. Lett.* **12**(5) (1981) 244–250.
[41] CROCHEMORE, M., Transducers and repetitions, *Theoret. Comput. Sci.* **45** (1986) 63–86.
[42] CROCHEMORE, M., String matching with constraints, Tech. Report 88-5, Dept. de Mathématique et Informatique, Univ. de Paris-Nord, 1988.
[43] CROCHEMORE, M. and D. PERRIN, Two-way pattern matching, Tech. Report, Univ. de Paris, 1989.
[44] DAVIES, G. and S. BOWSHER, Algorithms for pattern matching, *Software—Practice and Experience* **16**(6) (1986) 575–601.
[45] DIETZFELBINGER, M., A. KARLIN, K. MEHLHORN, F. MEYER AUF DER HEIDE, H. ROHNERT and R.E. TARJAN, Dynamic perfect hashing: upper and lower bounds, in: *Proc. 29th Ann. IEEE Symp. on Foundations of Computer Science* (1988) 524–531.
[46] DUVAL, J.P., Factorizing words over an ordered alphabet, *J. Algorithms* **4** (1983) 363–381.
[47] EILAN-TZOREFF, T. and U. VISHKIN, Matching patterns in strings subject to multi-linear transformations, *Theoret. Comput. Sci.* **60**(3) (1988) 231–254.
[48] FARBER, D.J., R.E. GRISWOLD and I.P. POLONSKY, SNOBOL, a string manipulation language, *J. ACM* **11**(1) (1964) 21–30.
[49] FISCHER, M.J. and M.S. PATERSON, String matching and other products, in: R.M. Karp, ed., *Complexity of Computation*, SIAM–AMS Proceedings, Vol. 7 (Amer. Mathematical Soc., Providence, RI, 1974) 113–125.
[50] FREDMAN, M.L., On computing the length of longest increasing subsequences, *Discrete Math.* **11**(1) (1975) 29–35.
[51] FREDMAN, M.L., J. KOMLOS and E. SZEMEREDI, Storing a sparse table with O(1) worst case access time, *J. ACM* **31**(3) (1984) 538–544.
[52] GALIL, Z., Two fast simulations which imply some fast string matching and palindrome recognition algorithms, *Inform. Process. Lett.* **4** (1976) 85–87.
[53] GALIL, Z., On improving the worst case running time of the Boyer–Moore string matching algorithm, *Comm. ACM* **22**(9) (1979) 505–508.
[54] GALIL, Z., String matching in real time, *J. ACM* **28**(1) (1981) 134–149.
[55] GALIL, Z. and R. GIANCARLO, Data structures and algorithms for approximate string matching, *J. Complexity* **4**(1) (1988) 33–72.
[56] GALIL, Z. and J.I. SEIFERAS, A linear-time on-line recognition algorithm for 'Palstar', *J. ACM* **25** (1978) 102–111.
[57] GALIL, Z. and J.I. SEIFERAS, Saving space in fast string matching, *SIAM J. Comput.* **9**(2) (1980) 417–438.
[58] GALIL, Z. and J. SEIFERAS, Time-space-optimal string matching, *J. Comput. System Sci.* **26** (1983) 280–294.
[59] GAREY, M.R. and D.S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, San Francisco, CA, 1979).
[60] GRISWOLD, R.E., J.F. POAGE and I.P. POLONSKY, *The SNOBOL4 Programming Language* (Prentice Hall, Englewood Cliffs, NJ, 2nd ed., 1971).
[61] GUIBAS, L.J. and A.M. ODLYZKO, A new proof of the linearity of the Boyer–Moore string searching algorithm, *SIAM J. Comput.* **9**(4) (1980) 672–682.
[62] GUIBAS, L.J. and A.M. ODLYZKO, Periods in strings, *J. Combin. Theory Ser. A* **30** (1981) 19–42.
[63] GUIBAS, L.J. and A.M. ODLYZKO, String overlaps, pattern matching, and nontransitive games, *J. Combin. Theory Ser. A* **30**(2) (1981) 183–208.
[64] HALL, P.A.V. and G.R. DOWLING, Approximate string matching, *ACM Comput. Surveys* **12**(4) (1980) 381–402.
[65] HARTMANIS, J. On the succinctness of different representations of languages, *SIAM J. Comput.* **9**(1) (1980) 114–120.

[66] HIRSCHBERG, D.S., A linear space algorithm for computing maximal common subsequences, *Comm. ACM* **18**(6) (1975) 341–343.
[67] HIRSCHBERG, D.S., Algorithms for the longest common subsequence problem, *J. ACM* **24**(4) (1977) 664–675.
[68] HIRSCHBERG, D.S., An information-theoretic lower bound for the longest common subsequence problem, *Inform. Process. Lett.* **7** (1978) 40–41.
[69] HOFFMAN, C.W. and M.J. O'DONNELL, Pattern matching in trees, *J. ACM* **29**(1) (1982) 68–95.
[70] HOPCROFT, J.E. and J.D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley, Reading, MA, 1979).
[71] HORSPOOL, R.N., Practical fast searching in strings, *Software—Practice and Experience* **10**(6) (1980) 501–506.
[72] HUET, G. and D.C. OPPEN, Equations and rewrite rules, in: R. Book, ed., *Formal Language Theory, Perspectives and Open Problems* (Academic Press, New York, 1980) 349–393.
[73] HUME, A.G., A tale of two greps, *Software—Practice and Experience* **18**(11) (1988) 1063–1072.
[74] HUNT, J.W. and M.D. MCILROY, An algorithm for differential file comparison, Computing Science Tech. Report 41, AT&T Bell Laboratories, Murray Hill, NJ, 1976.
[75] HUNT, J.W. and T.G. SZYMANSKI, A fast algorithm for computing longest common subsequences, *Comm. ACM* **20**(5) (1977) 350–353.
[76] IVANOV, A.G., Recognition of an approximate occurrence of words on a Turing machine in real time, *Math. USSR-Izv.* **24**(3) (1985) 479–522.
[77] KARP, R.M., R.E. MILLER, and A.L. ROSENBERG, Rapid identification of repeated patterns in strings, trees, and arrays, in: *Proc. 4th ACM Symp. on Theory of Computing* (1972) 125–136.
[78] KARP, R.M. and M.O. RABIN, Efficient randomized pattern-matching algorithms, *IBM J. Res. Develop.* **31**(2) (1987) 249–260.
[79] KLEENE, S.C., Representation of events in nerve nets and finite automata, in: C.E. Shannon and J. McCarthy, eds., *Automata Studies* (Princeton Univ. Press, Princeton, NJ, 1956) 3–42.
[80] KNUTH, D.E., J.H. MORRIS and V.R. PRATT, Fast pattern matching in strings, *SIAM J. Comput.* **6**(2) (1977) 323–350.
[81] LANDAU, G.M. and U. VISHKIN, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in: *Proc. 18th ACM Symp. on Theory of Computing* (1986) 220–230.
[82] LANDAU, G.M. and U. VISHKIN, Efficient string matching with k mismatches, *Theoret. Comput. Sci.* **43** (1986) 239–249.
[83] LANDAU, G.M. and U. VISHKIN, Fast string matching with k differences, *J. Comput. System Sci.* **37**(1) (1988) 63–78.
[84] LESK, M.E., LEX—a lexical analyzer generator, Computing Science Tech. Report 39, Bell Laboratories, Murray Hill, NJ, 1975.
[85] LI, M. and Y. YESHA, String-matching cannot be done by a two-head one-way deterministic finite automata, *Inform. Process. Lett.* **22** (1986) 231–236.
[86] LIU, K.-C., On string pattern matching: a new model with a polynomial time algorithm, *SIAM J. Comput.* **10**(1) (1981) 118–140.
[87] LOWRANCE, R. and R.A. WAGNER, An extension of the string-to-string correction problem, *J. ACM* **22** (1975) 177–183.
[88] LYNDON, R.C. and M.P. SCHÜTZENBERGER, The equation $a^M = b^N c^P$ in a free group, *Michigan Math. J.* **9** (1962) 289–298.
[89] MAIN, M.G. and R.J. LORENTZ, An $O(n \log n)$ algorithm for finding all repetitions in a string, *J. Algorithms* **5**(3) (1984) 422–432.
[90] MAIN, M.G. and R.J. LORENTZ, Linear time recognition of square-free strings, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985).
[91] MAJSTER, M.E. and A. REISNER, Efficient on-line construction and correction of position trees, *SIAM J. Comput.* **9**(4) (1980) 785–807.
[92] MANACHER, G., A new linear-time on-line algorithm for finding the smallest initial palindrome of a string, *J. ACM* **22** (1975) 346–351.
[93] MASEK, W.J. and M.S. PATERSON, A faster algorithm for computing string-edit distances, *J. Comput. System Sci.* **20**(1) (1980) 18–31.

[94] MCCULLOCH, W.S. and W. PITTS, A logical calculus of the ideas immanent in nervous activity, *Bull. Math. Biophysics* **5** (1943) 115–133.
[95] MCCREIGHT, E.M., A space-economical suffix tree construction algorithm, *J. ACM* **23**(2) (1976) 262–272.
[96] MCILROY, M.D., ed., *UNIX Time-Sharing System Programmer's Manual, Vol. I* (AT&T Bell Laboratories, Murray Hill, NJ, 9th ed., 1986).
[97] MCNAUGHTON, R. and H. YAMADA, Regular expressions and state graphs for automata, *IRE Trans. Electron. Comput.* **9**(1) (1960) 39–47.
[98] MEHLHORN, K., *Data Structures and Algorithms 1: Sorting and Searching* (Springer, Berlin, 1984).
[99] MEYER, A.R., and M.J. FISCHER, Economy of description by automata, grammars, and formal systems, in: *Proc. IEEE Symp. on Switching and Automata Theory* (1971) 188–190.
[100] MILLER, W., *A Software Tools Sampler* (Prentice Hall, Englewood Cliffs, NJ, 1987).
[101] MILLER, W., and E.W. MYERS, A file comparison program, *Software—Practice and Experience* **15**(11) (1985) 1025–1040.
[102] MILLER, W., and E.W. MYERS, Sequence comparison with concave weighting functions, *Bull. Math. Biol.* **50**(2) (1988) 97–120.
[103] MORRISON, D.R., PATRICIA—Practical algorithm to retrieve information coded in alphanumeric, *J. ACM* **15**(4) (1968) 514–534.
[104] MYERS, E.W., An $O(ND)$ difference algorithm and its variations, *Algorithmica* **1** (1986) 251–266.
[105] MYERS, E.W., A four Russians algorithm for regular expression pattern matching, Tech. Report 88-34, Dept. of Computer Science, Univ. of Arizona, Tucson, AZ, 1988.
[106] MYERS, E.W. and W. MILLER, Approximate matching of regular expressions, *Bull. Math. Biol.* **51**(1) (1989) 5–37.
[107] NAKATSU, N., Y. KAMBAYASHI and S. YAJIMA, A longest common subsequence algorithm suitable for similar text strings, *Acta Inform.* **18** (1982) 171–179.
[108] NEEDLEMAN, S.B. and C.D. WUNSCH, A general method applicable to the search for similarities in the amino acid sequences of two proteins, *J. Molecular Biol.* **48** (1970) 443–453.
[109] PENNELLO, T.J., Very fast LR parsing, *SIGPLAN Notices* **21**(7) (1986) 145–150.
[110] PERRIN, D., Finite automata, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (North-Holland, Amsterdam, 1990) 1–57.
[111] PINTER, R.Y., Efficient string matching with don't-care patterns, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985).
[112] RABIN, M.-O., Discovering repetitions in strings, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985).
[113] RABIN, M.O. and D. SCOTT, Finite automata and their decision problems, *IBM J. Res. Develop.* **3**(2) (1959) 114–125.
[114] RIVEST, R.L., On the worst-case behavior of string-searching algorithms, *SIAM J. Comput.* **6**(4) (1977) 669–674.
[115] RODEH, M., V.R. PRATT and S. EVEN, Linear algorithms for data compression via string matching, *J. ACM* **28**(1) (1981) 16–24.
[116] RYTTER, W., A correct preprocessing algorithm for Boyer–Moore string searching, *SIAM J. Comput.* **9**(3) (1980) 509–512.
[117] SALOMAA, K., Deterministic tree pushdown automata and monadic tree rewriting systems, *J. Comput. System Sci.* **37**(3) (1988) 367–394.
[118] SANKOFF, D. and J.B. KRUSKAL, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison* (Addison-Wesley, Reading, MA, 1983).
[119] SCHABACK, R., On the expected sublinearity of the Boyer–Moore algorithm, *SIAM J. Comput.* **17**(4) (1988) 648–658.
[120] SEDGEWICK, R., *Algorithms* (Addison-Wesley, Reading, MA, 1983).
[121] SEIFERAS, J. and Z. GALIL, Real-time recognition of substring repetition and reversal, *Math. Systems Theory* **11** (1977) 111–146.
[122] SELLERS, P.H., The theory and computation of evolutionary distances: pattern recognition, *J. Algorithms* **1** (1980) 359–373.

[123] SLISENKO, A.O., Detection of periodicities and string-matching in real time *J. Soviet Math.* **22**(3) (1983) 1316–1386 (originally published in 1980).

[124] SMIT, G. DE V., A comparison of three string matching algorithms, *Software—Practice and Experience* **12** (1982) 57–66.

[125] TAKAOKA, T., An on-line pattern matching algorithm, *Inform. Process. Lett.* **22** (1986) 329–330.

[126] TARJAN, R.E. and A.C. YAO, Storing a sparse table, *Comm. ACM* **22**(11) (1979) 606–611.

[127] THOMPSON, K., Regular expression search algorithm, *Comm. ACM* **11**(6) (1968) 419–422.

[128] THUE, A., Über unendliche Zeichenreihen, *Norske Videnskabers Selskabs Skrifter Mat.-Nat. Kl. (Kristiania)* **1** (1906) 1–22.

[129] THUE, A., Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen, *Norske Videnskabers Selskabs Skrifter Mat.-Nat. Kl. (Kristiania)* **7** (1912) 1–67.

[130] TICHY, W., The string-to-string correction problem with block moves, *ACM Trans. Comput. Systems* **2** (1984) 309–321.

[131] UKKONEN, E., Algorithms for approximate string matching, *Inform. and Control* **64** (1985) 100–118.

[132] UKKONEN, E., Finding approximate patterns in strings, *J. Algorithms* **6** (1985) 132–137.

[133] WAGNER, R., Order-*n* correction of regular languages, *Comm. ACM* **11**(6) (1974) 265–268.

[134] WAGNER, R. and M. FISCHER, The string-to-string correction problem, *J. ACM* **21**(1) (1974) 168–178.

[135] WAGNER, R. and J.I. SEIFERAS, Correcting counter-automaton-recognizable languages, *SIAM J. Comput.* **7**(3) (1978) 357–375.

[136] WATERMAN, M.S., General methods for sequence comparison, *Bull. Math. Biol.* **46**(4) (1984) 473–500.

[137] WEINER, P., Linear pattern matching algorithms, in: *Proc. 14th IEEE Symp. on Switching and Automata Theory* (1973) 1–11.

[138] WONG, C.K. and A.K. CHANDRA, Bounds for the string editing problem, *J. ACM* **23**(1) (1976) 13–16.

[139] YAO, A.C., The complexity of pattern matching for a random string, *SIAM J. Comput.* **8** (1979) 368–387.