# 4

# Extended string matching

## 4.1 Basic concepts

Up to now we have considered search patterns that are sequences of characters. However, in many cases one may be interested in a more sophisticated form of searching. The most complex patterns that we consider in this book are regular expressions, which are covered in Chapter 5. However, regular expression searching is costly in processing time and complex to program, so one should resort to it only if necessary. In many cases one needs far less flexibility, and the search problem can be solved more efficiently with much simpler algorithms.

We have designed this chapter on "extended strings" as a middle point between simple strings and regular expressions. We provide simple search algorithms for a number of enhancements over the basic string search, which can be solved more easily than general regular expressions. We focus on those used in text searching and computational biology applications.

We consider four extensions to the string search problem: classes of characters, bounded length gaps, optional characters, and repeatable characters. The first one allows specifying sets of characters at any pattern or text position. The second permits searching patterns containing bounded length gaps, which is of interest for protein searching (e.g., PROSITE patterns [Gus97, HBFB99]). The third allows certain characters to appear optionally in a pattern occurrence, and the last permits a given character to appear multiple times in an occurrence, which includes wild cards. We finally consider some limited multipattern search capabilities.

Different occurrences of a pattern may have different lengths, and there may be several occurrences starting or ending at the same text position. Among the several choices for reporting these occurrences, we choose to



https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

report all the initial or all the final occurrence positions, depending on what is more natural for each algorithm.

In this chapter we make heavy use of bit-parallel algorithms. With some extra work, other algorithms can be adapted to handle some extended patterns as well, but bit-parallel algorithms provide the maximum flexibility and in general the best performance. We show that **Shift-And** can be adapted by changing the mechanism to simulate a new nondeterministic automaton. **BNDM** can be adapted as well, although we will be faced with the problem that the pattern occurrences need not have the same length as the pattern, so it will be necessary to verify, each time we arrive at the beginning of a window, whether we have a real match or not.

All the techniques in this chapter can be plugged into the algorithms in Chapter 6 for approximate searching. Some can also be combined with regular expression searching (Chapter 5).

## 4.2 Classes of characters

### *4.2.1 Classes in the pattern*

Our simplest extension of string matching permits each pattern position to match a *set* of characters rather than a single character. The pattern is a sequence over $\wp(\Sigma)$, that is, $p = p_1 p_2 \ldots p_m$, where $p_j \subseteq \Sigma$. We say that $p' \in \Sigma^*$ is an occurrence of $p$ whenever $p'_j \in p_j$ for all $j \in 1 \ldots m$. A simple string is a particular case of this type of pattern.

It is usual to denote sets of characters by enumerating their components in square brackets, or by using ranges of characters when a total linear order is clear. For example, `"[Aa]nnual"` matches `"Annual"` and `"annual"`, while `"[0-9][0-9]/[0-9][0-9]/199[0-9]"` matches dates in the 1990s. We will use this notation throughout the chapter, as well as the symbol $\Sigma$ to denote a pattern position matching the whole alphabet.

Two simple extensions that can be expressed using classes of characters are (1) "don't care" symbols, which match any text character, corresponding to the class $\Sigma$; (2) case-insensitive searching, which corresponds to replacing each pattern character by a class formed from its uppercase and its lowercase version; for example, `"[Aa][Nn][Nn][Uu][Aa][Ll]"` matches the string `"annual"` in case-insensitive form.

Assume that we have a bit-parallel algorithm, such as **Shift-And** or **BNDM** (Chapter 2). The only connection between the pattern and the text is made at preprocessing time by building a table $B$, which for each character $c$ gives the bit mask of the pattern positions matching $c$. Now assume that $p$ is a sequence of classes of characters. The bit-parallel al-

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

gorithms can be used directly provided we change the preprocessing. We replace line 3 of **Shift-And** (Figure 2.6) by

**For** $j \in 1 \ldots m$ **Do**
  **For** $c \in p_j$ **Do** $B[c] \leftarrow B[c] \mid 0^{m-j}10^{j-1}$
**End of for**

or, alternatively, line 3 of **BNDM** (Figure 2.16) by

**For** $j \in 1 \ldots m$ **Do**
  **For** $c \in p_j$ **Do** $B[c] \leftarrow B[c] \mid 0^{j-1}10^{m-j}$
**End of for**

**Shift-Or** needs to reverse the bits and change "|" to "&". Figure 4.1 shows an example of the resulting mask $B$ for the **Shift-And**.

| $c$ | $B[c]$ |
|---|---|
| 0 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 1 | 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 |
| 2 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 3 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 4 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 5 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 6 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 7 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 8 | 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |

| $c$ | $B[c]$ |
|---|---|
| 9 | 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| A | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 |
| a | 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 |
| f | 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 |
| l | 0 0 0 0 0 0 0 0 0 1 0 0 0 0 |
| n | 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 |
| o | 0 0 0 0 0 0 1 0 0 0 0 0 0 0 |
| s | 0 0 0 0 0 0 0 0 1 0 0 0 0 0 |
| _ | 0 0 0 0 1 0 0 1 0 0 0 0 0 0 |

Fig. 4.1. The resulting mask $B$ for the pattern `"[Aa]nnals_of_199[0-9]"`.

Non-bit-parallel algorithms can be extended to handle classes of characters too, but none of them provide the same combination of simplicity and performance robustness. Let us examine first the **Horspool** algorithm (Section 2.3.2). We need to change, in Figure 2.12, line 3 in the preprocessing and line 8 in the search. However, the performance of **Horspool** degrades rapidly, especially if there is a large class near the end of the pattern. This is because the shifts for all the characters contained in the large class will be short. Thus its performance is extremely sensitive to the number, size, and position of the classes.

Now consider **BDM** (Section 2.4.1). No efficient algorithm is known to

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

extend the deterministic suffix automaton to handle classes of characters [NR00]. The same is true for the **BOM** algorithm of Section 2.4.3.

From the classical algorithms, the extension that performs best is the classical **Boyer-Moore** algorithm (Section 2.3.1). However, it is complex to implement and does not perform as well as **BNDM** [NR00].

Performance of **Shift-And/Shift-Or** is unaffected by the use of classes of characters. However, it is inferior to that of **BNDM** in most cases. But **BNDM** is affected because it is more likely to find occurrences of pattern factors in the window. A rough analysis is as follows: If $S$ is the average size of a class, then the result is the same as if we had an alphabet of size $|\Sigma|/S$, and hence the average complexity of **BNDM** becomes $O(n \log_{|\Sigma|/S}(m)/m)$.

Just as **BNDM** is better than **Horspool** with smaller alphabets, it is more resistant than **Horspool** to the size and number of classes in the pattern. When the classes become too numerous or too large, it may be better to switch to **Shift-Or**, which is slightly faster than **Shift-And**. However, in the extensions that we consider next **Shift-Or** is not faster, and **Shift-And** is preferable because it is more intuitive.

### *4.2.2 Classes in the text*

In computational biology applications there may be uncertainty on some text characters; that is, one knows that a given text position holds some character in a given set, but cannot tell which one. This situation is modeled by allowing classes of characters in the text. It is normally represented by using new character codes that are known to represent given sets of "normal" character codes.

Formally, the text is a sequence over $\wp(\Sigma)$, that is, $T = t_1 t_2 \dots t_n$, where $t_i \subseteq \Sigma$. The pattern is said to occur at text position $t_{i+1} \dots t_{i+m}$ if $p_j \cap t_{i+j} \neq \emptyset$ for all $j \in 1 \dots m$.

Bit-parallelism gives a simple way to deal with this. Let us say that character code $c$ represents the set $\{c_1, c_2, \dots, c_k\}$. Then, *after* building the mask $B$ of the normal characters in the preprocessing of either **Shift-And** or **BNDM**, we add for each such $c$

$B[c] \leftarrow 0^m$
**For** $i \in 1 \dots k$ **Do** $B[c] \leftarrow B[c] \mid B[c_i]$

which makes $c$ match every pattern position that matches some $c_i$.

This can be extended to permit special characters that are sets of other special characters, as long as they are processed in the correct order. More-

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

over, it can be combined with classes of characters in the pattern, which are dealt with when the table $B$ of the normal characters is built.

On small alphabets, such as that of DNA sequences, an interesting choice is to extend the character set to $\Sigma' = \{0 \ldots 2^{|\Sigma|} - 1\}$ and represent the set using bit-parallelism. The new alphabet is formed by bit masks of length $|\Sigma|$, where the $i$-th bit indicates the presence in the set of the $i$-th character. For example, if the alphabet is $\{A, G, C, T\}$, then single characters will be represented by $A = 0001$, $G = 0010$, $C = 0100$, and $T = 1000$, and classes will be represented by, for example, $\{A, C\} = 0101$.

Under this representation we need to build a different table $B'$ that ranges over the integers $\{0 \ldots 2^{|\Sigma|} - 1\}$. Assume for simplicity that $\Sigma = \{0 \ldots |\Sigma| - 1\}$. The construction of $B'$, given $B$, is as follows:

$B'[0] \leftarrow 0^m$
**For** $c \in 0 \ldots |\Sigma| - 1$ **Do**
  **For** $j \in 0 \ldots 2^c - 1$ **Do** $B'[2^c + j] \leftarrow B[c] \mid B'[j]$
**End of for**

It takes $O(2^{|\Sigma|})$ time. With DNA, for example, the table $B'$ has just 16 entries. The search process is unaltered: We simply use $B'$ instead of $B$.

## 4.3 Bounded length gaps

An important case of protein searching is that of PROSITE patterns [Gus97, HBFB99]. A PROSITE pattern contains classes of characters and bounded length gaps, which match any string whose length is between given bounds. In the notation of PROSITE, pattern characters or classes are separated by hyphens and $x(a, b)$ denotes a gap of length between $a$ and $b$. Also, $x(a) = x(a, a)$ and $x = x(1)$, which is equivalent to the class $\Sigma$. For example, the pattern $a - b - c - x(1, 3) - d - e$ matches `"abcfde"` and `"abcfddde"`, but not `"abcffffde"`. Although we focus on the concrete case of PROSITE patterns in this chapter, the algorithms can handle other types of pattern with bounded length gaps.

Figure 4.2 shows an NFA for the pattern $a-b-c-x(1,3)-d-e$. Between the characters `"c"` and `"d"` we have inserted three transitions that can be followed by any character, which corresponds to the maximum length of the gap. Two $\varepsilon$-transitions leave the state where `"abc"` has been recognized and skip one and two subsequent edges, respectively. This skips one to three text characters before finding the `"de"` at the end of the pattern. The initial self-loop allows the match to begin at any text position.

Let $m$ be the number of symbols in the pattern, each symbol being a class

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

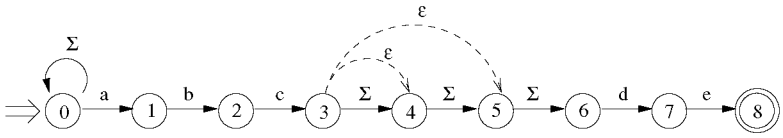


Fig. 4.2. A nondeterministic automaton for the pattern $a - b - c - x(1,3) - d - e$. Dashed arrows represent $\varepsilon$-transitions, which can be followed without consuming any input.

of characters or a gap specification of the form $x(a,b)$. Also let $\ell min$ and $\ell max$ be the minimum and maximum lengths of a pattern occurrence. Both can be obtained from the pattern in $O(m)$ time by adding 1 for each class of characters and adding $a$ for the minimum and $b$ for the maximum for each gap specification $x(a,b)$. Finally, let $L$ be the number of states of the corresponding NFA, not including the first state. It is not hard to see that $L = \ell max$. In our example, $m = 6$, $\ell min = 6$, and $\ell max = L = 8$.

We now describe two bit-parallel algorithms presented in [NR01b] which are able to find patterns quickly patterns with gaps (PROSITE in particular). They extend **Shift-And** and **BNDM**.

### *4.3.1 Extending Shift-And*

We augment the representation of **Shift-And** by adding the $\varepsilon$-transitions. We call "gap-initial" states those states $i$ from which an $\varepsilon$-transition leaves. For each gap-initial state $i$ corresponding to a gap $x(a,b)$, we define its "gap-final" state to be $(i + b - a + 1)$, that is, the one *following* the last state reached by an $\varepsilon$-transition leaving $i$. In Figure 4.2, we have one gap-initial state (3) and one gap-final state (6).

We create a bit mask $I$ that has 1 in the gap-initial states and another mask $F$ that has 1 in the gap-final states. In Figure 4.2, the corresponding $I$ and $F$ masks are 00000100 and 00100000, respectively. After performing the normal **Shift-And** step, we simulate all the $\varepsilon$-moves with the operation

$$D \leftarrow D \mid ((F - (D \,\&\, I)) \,\&\, \sim F)$$

The rationale is as follows. $D \,\&\, I$ isolates the *active* gap-initial states. Subtracting this from $F$ has two possible outcomes for each gap-initial state $i$. First, if $i$ is active, the result will have 1 in all the states from $i$ to $(i + b - a)$, successfully propagating the active state $i$ to the desired target states. Second, if $i$ is inactive, the outcome will have 1 only in state $(i + b - a + 1)$. This undesired 1 is removed by operating on the result with "$\& \sim F$". Once the propagation has been done, we OR the result with the already active

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

states in $D$. Note that the propagations of different gaps do not interfere with one another, since all the subtractions have a local effect.

---

**Gaps-Shift-And** $(p = p_1 p_2 \dots p_m,\ T = t_1 t_2 \dots t_n)$

1. `Preprocessing`
2. $L \leftarrow$ maximum length of an occurrence
3. **For** $c \in \Sigma$ **Do** $B[c] \leftarrow 0^L$
4. $I \leftarrow 0^L,\ F \leftarrow 0^L$
5. $i \leftarrow 0$
6. **For** $j \in 1 \dots m$ **Do**
7. **If** $p_j$ is of the form $x(a,b)$ **Then**
8. $I \leftarrow I \mid 0^{L-i}10^{i-1}$
9. $F \leftarrow F \mid 0^{L-(i+b-a)-1}10^{i+b-a}$
10. **For** $c \in \Sigma$ **Do** $B[c] \leftarrow B[c] \mid 0^{L-i-b}1^b0^i$
11. $i \leftarrow i + b$
12. **Else** /* $p_j$ is a class of characters */
13. **For** $c \in p_j$ **Do** $B[c] \leftarrow B[c] \mid 0^{L-i-1}10^i$
14. $i \leftarrow i + 1$
15. **End of if**
16. **End of for**
17. `Searching`
18. $D \leftarrow 0^L$
19. **For** $pos \in 1 \dots n$ **Do**
20. $D \leftarrow ((D << 1) \mid 0^{L-1}1)\ \&\ B[t_{pos}]$
21. $D \leftarrow D \mid ((F - (D\ \&\ I))\ \&\ \sim F)$
22. **If** $D\ \&\ 10^{L-1} \neq 0^L$ **Then** report an occurrence ending at $pos$
23. **End of for**

---

Fig. 4.3. The extension of **Shift-And** to handle PROSITE expressions.

Figure 4.3 shows the complete algorithm. For simplicity we assume that there are no gaps at the beginning or at the end of the pattern and that consecutive gaps have been merged into one. The preprocessing takes $O(m|\Sigma|)$ time, while the scanning needs $O(n)$ time. If $\ell max > w$, however, we need several machine words for the simulation, and it then takes $O(n\lceil \ell max/w \rceil)$ time.

**Example of Gaps-Shift-And** We search for the pattern $a-b-c-x(1,3)-d-e$ in the text `"abcabcffdee"`.

$B =$

| | |
|---|---|
| a | 0 0 1 1 1 0 0 1 |
| b | 0 0 1 1 1 0 1 0 |
| c | 0 0 1 1 1 1 0 0 |
| d | 0 1 1 1 1 0 0 0 |
| e | 1 0 1 1 1 0 0 0 |
| * | 0 0 1 1 1 0 0 0 |

$I = 0\,0\,0\,0\,0\,1\,0\,0$
$F = 0\,0\,1\,0\,0\,0\,0\,0$
$D = 0\,0\,0\,0\,0\,0\,0\,0$

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

1. Reading `a` 0 0 1 1 1 0 0 1
   $D =$ 0 0 0 0 0 0 0 1
   We now apply the propagation formula on $D$. The result is $((F - (D \ \& \ I)) \ \& \ \sim F) = ((00100000 - 00000000) \ \& \ 11011111) = 00000000$, and hence $D$ does not change. We do not mention again the propagation formula unless it has an effect on $D$.

2. Reading `b` 0 0 1 1 1 0 1 0
   $D =$ 0 0 0 0 0 0 1 0

3. Reading `c` 0 0 1 1 1 1 0 0
   $D =$ 0 0 0 0 0 1 0 0
   At this point the $\varepsilon$-transitions take effect: $((F - (D \ \& \ I)) \ \& \sim F)$ yields $((00100000 - 00000100) \ \& \ 11011111) = 00011100$, where states 4 and 5 have been activated. The new $D$ value is
   $D = 0\,0\,0\,1\,1\,1\,0\,0.$

4. Reading `a` 0 0 1 1 1 0 0 1
   $D =$ 0 0 1 1 1 0 0 1

5. Reading `b` 0 0 1 1 1 0 1 0
   $D =$ 0 0 1 1 0 0 1 0

6. Reading `c` 0 0 1 1 1 1 0 0
   $D =$ 0 0 1 0 0 1 0 0
   The propagation formula takes effect again and produces
   $D = 0\,0\,1\,1\,1\,1\,0\,0.$

7. Reading `f` 0 0 1 1 1 0 0 0
   $D =$ 0 0 1 1 1 0 0 0

8. Reading `f` 0 0 1 1 1 0 0 0
   $D =$ 0 0 1 1 0 0 0 0

9. Reading `d` 0 1 1 1 1 0 0 0
   $D =$ 0 1 1 0 0 0 0 0

10. Reading `e` 1 0 1 1 1 0 0 0
    $D =$ 1 0 0 0 0 0 0 0
    The last bit of $D$ is set, so we mark an occurrence. The gap has matched the text `"ff"`.

11. Reading `e` 1 0 1 1 1 0 0 0
    $D =$ 0 0 0 0 0 0 0 0

### *4.3.2 Extending BNDM*

We now try to extend **BNDM** (Section 2.4.2) to handle patterns with gaps. To recognize all the reverse factors of the pattern, we use the same automaton of Figure 4.2 on the reversed pattern, but without the initial self-loop, and we consider that all the states are active at the beginning. Figure 4.4 shows the automaton for the pattern $a - b - c - x(1,3) - d - e$. A string read by this automaton is a factor of the pattern as long as there exists at least one active state. Note that now the arrows depart from the state next to `"d"`, but the effect is the same as before.

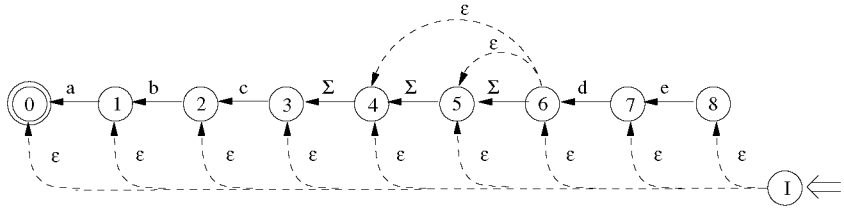


Fig. 4.4. A nondeterministic automaton to recognize all the reversed factors of the PROSITE pattern $a - b - c - x(1,3) - d - e$.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

The bit-parallel simulation of this automaton is similar to that of the forward automaton. The only modifications are (*a*) we build it on the reversed pattern; (*b*) the bit mask $D$ that registers the state of the search has to be initialized with $D = 1^L$ to represent the initial $\varepsilon$-transitions; and (*c*) we do not OR $D$ with $0^{L-1}1$ when we shift it, since there is no longer an initial loop.

The backward matching algorithm shifts a window of size $\ell min$ along the text. Inside each window, the algorithm traverses the text backwards trying to recognize a factor of the pattern. Each time the automaton reaches its final state we have recognized a pattern prefix and we store the window position in the variable *last*.

If the backward search inside the window fails before reaching the beginning of the window, then the search window is shifted to the beginning of the longest prefix recognized, as in **BNDM**.

If the beginning of the window is reached with the automaton still holding active states, then some factor of length $\ell min$ of the pattern has been recognized in the window. Unlike exact string matching, where all occurrences have the length of the pattern, reaching the beginning of the window here does not automatically imply that we have recognized the whole pattern. We need to verify a possible occurrence, which can be as long as $\ell max$, starting at the beginning of the window.

To carry out this verification, we read the characters again from the beginning of the window with the forward automaton of Figure 4.2, but without the initial self-loop. This makes the automaton *recognize* rather than *search for* the pattern. To simulate that automaton without the initial self-loop, we simply do

$$\begin{aligned} D &\leftarrow (D << 1) \ \& \ B[t_{pos}] \\ D &\leftarrow D \mid ((F - (D \ \& \ I)) \ \& \ \sim F) \end{aligned}$$

This forward verification ends when either (1) the automaton reaches its final state, in which case we have found the pattern; or (2) the automaton runs out of active states, in which case there is no pattern occurrence starting at the window. Since there is no initial loop, the forward verification surely finishes after reading at most $\ell max$ text characters. We then shift the search window to the position of the last pattern prefix recognized and resume the search.

Figure 4.5 shows the complete algorithm. Its worst-case complexity is $O(n \times \ell max)$, which is poor in theory. In particular, let us consider the

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

```
Gaps-BNDM (p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n)
    Preprocessing
        L ← maximum length of an occurrence
        ℓmin ← minimum length of an occurrence
        For c ∈ Σ Do B[c] ← 0^L
        I ← 0^L, F ← 0^L
        i ← 0
        For j ∈ 1...m Do
            If p_j is of the form x(a,b) Then
                I ← I | 0^{i+b} 10^{L-(i+b)-1}
                F ← F | 0^{i+a-1} 10^{L-(i+a)}
                For c ∈ Σ Do B[c] ← B[c] | 0^i 1^b 0^{L-i-b}
                i ← i + b
            Else /* p_j is a class of characters */
                For c ∈ p_j Do B[c] ← B[c] | 0^i 10^{L-i-1}
                i ← i + 1
            End of if
        End of for
    Searching
        pos ← 0
        While pos ≤ n − ℓmin Do
            j ← ℓmin, last ← ℓmin
            D ← 1^L
            While D ≠ 0^L AND j > 0 Do
                D ← D & B[t_{pos+j}]
                D ← D | ((F − (D & I)) & ∼F)
                j ← j − 1
                If D & 10^{L-1} ≠ 0^L Then /* prefix recognized */
                    If j > 0 Then last ← j
                    Else check a possible occurrence starting at pos
                End of if
                D ← D << 1
            End of while
            pos ← pos + last
        End of while
```

Fig. 4.5. The extension of **BNDM** to handle PROSITE expressions.

maximum gap length $G$ in the pattern. If $G \geq \ell min$, then *every* text window of length $\ell min$ *is* a factor of the pattern; so we will always traverse the whole window during the backward scan, for a minimum complexity of $O(n)$. Consequently, this approach should not be used when $G \geq \ell min$. It has been shown experimentally in [NR01b] that **Gaps-BNDM** is better than **Gaps-Shift-And** whenever $G + 1 < \ell min/2$.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

**Example of Gaps-BNDM** We search for the pattern $a-b-c-x(1,3)-d-e$ in the text `"abcabcffdee"`.

$B =$

| | |
|---|---|
| a | 1 0 0 1 1 1 0 0 |
| b | 0 1 0 1 1 1 0 0 |
| c | 0 0 1 1 1 1 0 0 |
| d | 0 0 0 1 1 1 1 0 |
| e | 0 0 0 1 1 1 0 1 |
| * | 0 0 0 1 1 1 0 0 |

$I = 0\,0\,0\,0\,0\,1\,0\,0$
$F = 0\,0\,1\,0\,0\,0\,0\,0$
$\ell min = 6,\ \ell max = 8$

1. `abcabc` `ffdee`
   $last \leftarrow 6$

   Reading `c` 0 0 1 1 1 1 0 0
   $D =$ 0 0 1 1 1 1 0 0

   The propagation mechanism does not introduce any new active states in $D$.

   Reading `b` 0 1 0 1 1 1 0 0
   $D =$ 0 1 0 1 1 0 0 0

   Reading `a` 1 0 0 1 1 1 0 0
   $D =$ 1 0 0 1 0 0 0 0

   The last bit of $D$ is activated and $j > 0$, so we set $last \leftarrow 3$.

   Reading `c` 0 0 1 1 1 1 0 0
   $D =$ 0 0 1 0 0 0 0 0

   Reading `b` 0 1 0 1 1 1 0 0
   $D =$ 0 1 0 0 0 0 0 0

   Reading `a` 1 0 0 1 1 1 0 0
   $D =$ 1 0 0 0 0 0 0 0

   The last bit of $D$ is active and $j = 0$, so we start a forward verification against the text `"abcabcff"`. The forward automaton finally dies without finding the pattern and we proceed to the next window, shifting by $last = 3$.

2. `abc` `abcffd` `ee`
   $last \leftarrow 6$

   Reading `d` 0 0 0 1 1 1 1 0
   $D =$ 0 0 0 1 1 1 1 0

   The propagation mechanism is activated, but it produces no effects.

   Reading `f` 0 0 0 1 1 1 0 0
   $D =$ 0 0 0 1 1 1 0 0

   The propagation mechanism is activated, but again it produces no effects.

   Reading `f` 0 0 0 1 1 1 0 0
   $D =$ 0 0 0 1 1 0 0 0

   Reading `c` 0 0 1 1 1 1 0 0
   $D =$ 0 0 1 1 0 0 0 0

   Reading `b` 0 1 0 1 1 1 0 0
   $D =$ 0 1 0 0 0 0 0 0

   Reading `c` 1 0 0 1 1 1 0 0
   $D =$ 1 0 0 0 0 0 0 0

   The last bit of $D$ is set and $j = 0$, so we perform a forward verification against the text `"abcffdee"`, which produces a match. Therefore, the current text position (4) is reported as the beginning of an occurrence.
   The window is shifted by $last = 6$ and we finish the search.

## 4.4 Optional characters

We now allow the possibility that some pattern positions may or may not appear in the text. We call these "optional characters" (or classes) and denote them by putting a question mark after the optional position. Consider the pattern `"abc?d?efg?h"`, which matches, for example, `"abefh"` and `"abdefgh"`. A nondeterministic automaton accepting this pattern is shown in Figure 4.6.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

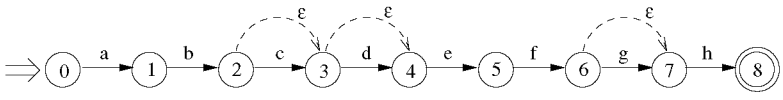


Fig. 4.6. A nondeterministic automaton accepting the pattern `"abc?d?efg?h"`.

As the figure shows, multiple consecutive optional characters could exist. The simplest solution, for when that does not happen, is to set up a bit mask $O$ with 1's in the optional positions (in our example, $O = 01001100$) and let the 1's in previous states of $D$ propagate to them. Hence, after the normal update to $D$ in, say, the **Shift-And** algorithm (i.e., after line 7 in Figure 2.6), we perform

$$D \leftarrow D \mid ((D << 1) \ \& \ O)$$

This solution works if we have read `"abcdef"` (then $D = 00100000$) and the next text character is `"h"`, since the above operation would convert $D$ into 01100000 before operating it against $B[\mathtt{h}] = 10000000$. However, it does not work if the text is `"abefgh"`, where both consecutive optional characters have been omitted.

A general solution needs to propagate each active state in $D$ so as to flood with 1's all the states ahead of it that correspond to optional characters. In our example, when $D$ is 00000010 we would like it to become 00001110 after the flooding.

This is achieved in [Nav01b] with a mechanism resembling that of Section 4.3. Three masks, $A$, $I$, and $F$, mark the boundaries of *blocks* of consecutive optional characters. Each block starts at the position *before* the first optional character in the sequence and finishes at the position of the last optional character. For example, in Figure 4.6 the first block starts at position 2 and ends at position 4. The $i$-th bit of $A$ is set if position $i$ in $p$ is optional, that of $I$ is set if $i$ is the position *preceding* the first optional character of a block, and that of $F$ is set if $i$ is the position of the last optional character of a block. In our example, $A = 01001100$, $I = 00100010$, and $F = 01001000$. After performing the normal transition on $D$, we do the following

$$\begin{array}{lcl} Df & \leftarrow & D \mid F \\ D & \leftarrow & D \mid (A \ \& \ ((\sim (Df - I)) \ \wedge \ Df)) \end{array}$$

The first line sets the positions finishing blocks in $D$ to 1. In the second line we add some active states to $D$. Since the states to add are AND-ed with $A$, let us consider what happens inside a specific block. We want the

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

first 1 counting from the right to flood all the block bits to its left. We subtract $I$ from $Df$, which is equivalent to subtracting 1 at each block. This subtraction cannot propagate outside the block because there is a 1 coming from "| $F$" in $Df$ at the highest bit of the block. The effect of the subtraction is that all the bits until the first 1 (counting from the right) are reversed (e.g., $1000000 - 1 = 0111111$) and the rest are unchanged. In general, $b_x b_{x-1} \ldots b_{x-y} 10^z - 1 = b_x b_{x-1} \ldots b_{x-y} 01^z$. When this is reversed by the "$\sim$" operation we get $\sim b_x \ \sim b_{x-1} \ldots \ \sim b_{x-y} 10^z$. Finally, when this is XOR-ed with the same $Df = b_x b_{x-1} \ldots b_{x-y} 10^z$ we get $1^{x-y+1} 0^{z+1}$.

This gives the effect we wanted: The first 1 flooded all the bits to the left. The 1 itself has been converted to 0, but it is restored when the result is OR-ed with the original $D$. This works even if the last active state in the optional block is the leftmost bit of the block. Note that it is necessary to AND with $A$ at the end to avoid propagating the XOR outside the block. We will see a combined example at the end of Section 4.5.

Note that optional characters cannot be expressed as gaps, since they can appear consecutively and they do not necessarily match with arbitrary characters. On the other hand, bounded length gaps *can* be expressed using optional characters; for example, $a - b - c - x(1,3) - d - e$ is equivalent to `"abc`$\Sigma\Sigma?\Sigma?$`de"`. However, the formula for the case of bounded length gaps is simpler and hence faster.

## 4.5 Wild cards and repeatable characters

"*Wild card*" is a term used to refer to a pattern position that matches an arbitrarily long text string, and it is usually denoted with a star. For example, `"ann*al"` matches the texts `"annal"`, `"annual"`, and `"annals of biological"`. We are *not* using this notation because we prefer a more general one.

A wild card is a particular class of a more general feature called a "repeatable character." A repeatable character is a pattern position that can appear zero or more times in the text. We denote it with the character or class of characters followed by an asterisk; for example, `AC*TCA` matches `ATCA`, as well as `ACCTCA` and `ACCCCCCCTCA`. Another example is `"[a-zA-Z_][a-zA-Z_0-9]*"`, which matches valid variable names in most programming languages. Under this definition, a wild card is expressed as `"`$\Sigma$`*"`. The algorithms for repeatable characters are not slower nor harder to program than those for simple wild cards. Following regular expression notations, we also denote $c+ = cc*$.

Figure 4.7 shows a possible automaton for the pattern `"abc+def*gh"`.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

However, it is difficult to extend to the case where consecutive characters with stars appear, for example, `"abc+def*g*h"`.

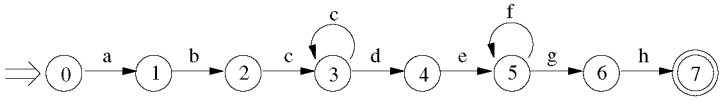


Fig. 4.7. A nondeterministic automaton accepting the pattern `"abc+def*gh"`. The mechanism cannot be extended to consecutive stars.

A general solution that permits consecutive stars is based on the identity $c* = c+?$. We simulate directly the "+" and express the "$*$" operator in terms of "+" and "?". Figure 4.8 shows the automaton we use for `"abc+def*gh"`. Hence, to deal with "$*$" we need to deal with repeatable and optional characters.

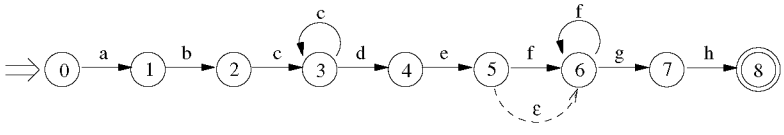


Fig. 4.8. A nondeterministic automaton accepting the pattern `"abc+def*gh"`. It can deal with consecutive stars.

Let $m$ be the pattern length, counting both normal characters and the three special symbols. The minimum length of an occurrence, $\ell min$, is computed in $O(m)$ time as the number of normal characters in the pattern excluding those affected by "?" and "$*$" operators. On the other hand, the maximum length of an occurrence is unbounded when there are repeatable characters. Finally, let $L$ be the number of states in the NFA (excluding the first one) computed as the number of normal characters in $p$.

For bit-parallel simulation of the operator "+" we need a table $S[c]$ that for each character $c$ tells which pattern positions can remain active when we read the character $c$. In Figure 4.8, $S[\mathtt{c}] = 00000100$ and $S[\mathtt{f}] = 00100000$. A complete simulation step permitting optional and repeatable characters after reading text character $t_{pos}$ is as follows:

$$\begin{aligned} D &\leftarrow ((D << 1) \mid 0^{L-1}1) \ \& \ B[t_{pos}]) \ \mid \ (D \ \& \ S[t_{pos}]) \\ Df &\leftarrow D \mid F \\ D &\leftarrow D \mid (A \ \& \ ((\sim (Df - I)) \ \wedge \ Df)) \end{aligned}$$

The complete code is quite similar to that of patterns with gaps detailed in Section 4.3, the only change being in the simulation of a single step of the NFA. We present extended versions of **Shift-And** and of **BNDM**.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

Some experimental results are presented in [Nav01b] regarding the use of optional and repeatable characters. It is shown that **Extended-BNDM** works better in most cases than **Extended-Shift-And**. The latter choice should be considered only when $\ell min \leq 3$ or when there are large repeatable classes of characters.

### 4.5.1 Extended Shift-And

Figure 4.9 shows pseudo-code for the **Shift-And** extension. It includes the necessary preprocessing of the pattern to deal with the symbols `"+"`, `"*"`, and `"?"`. The code assumes that there are no optional or repeatable characters at the beginning or at the end of the pattern. It is not hard to augment the code to handle classes of characters.

Consider the **Extended-Shift-And** algorithm preprocessing. The preprocessing has two parts. Lines 2–17 build the mask $A$ and the tables $S$ and $B$, where $S$ stores information about repeatable characters and $A$ stores information about optional characters. The operator "$*$" is treated exactly like "+" followed by "?". Lines 18–29 build the $I$ and $F$ masks from $A$, by the simple mechanism of detecting in line 21 whether the current active bit of $A$ belongs to a new block or not, and, if not, "moving" the bit of $F$ that signals its end. The preprocessing takes $O(m + |\Sigma|)$ time and the search $O(n\lceil L/w \rceil)$ time.

The search code is simple compared to the preprocessing. It applies the formula to deal with optional and repeatable characters.

**Example of Extended-Shift-And** We search for the ending position of occurrences of the pattern `"ab?c*de+f"` in the text `"acccdfabdeeef"`. We have $m = 9$ and $L = 6$. For each character we show the effect of the three lines of the processing done on $D$ and $Df$.

| $c$ | $B$ | $S$ |
|---|---|---|
| a | 0 0 0 0 0 1 | 0 0 0 0 0 0 |
| b | 0 0 0 0 1 0 | 0 0 0 0 0 0 |
| c | 0 0 0 1 0 0 | 0 0 0 1 0 0 |
| d | 0 0 1 0 0 0 | 0 0 0 0 0 0 |
| e | 0 1 0 0 0 0 | 0 1 0 0 0 0 |
| f | 1 0 0 0 0 0 | 0 0 0 0 0 0 |
| * | 0 0 0 0 0 0 | 0 0 0 0 0 0 |

$$A = 0\,0\,0\,1\,1\,0$$
$$I = 0\,0\,0\,0\,0\,1$$
$$F = 0\,0\,0\,1\,0\,0$$
$$D = 0\,0\,0\,0\,0\,0$$

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

**Extended-Shift-And** ($p = p_1 p_2 \ldots p_m$, $T = t_1 t_2 \ldots t_n$)

```
1.    Preprocessing
2.        L ← number of normal characters in p
3.        A ← 0^L /* build B, S and A */
4.        For c ∈ Σ Do B[c] ← 0^L, S[c] ← 0^L
5.        i ← −1
6.        For j ∈ 1...m Do
7.            If p_j = "+" Then S[lastc] ← S[lastc] | 0^{L−i−1}10^i
8.            Else If p_j = "?" Then A ← A | 0^{L−i−1}10^i
9.            Else If p_j = "*" Then
10.               S[lastc] ← S[lastc] | 0^{L−i−1}10^i
11.               A ← A | 0^{L−i−1}10^i
12.           Else /* p_j is a character */
13.               lastc ← p_j
14.               i ← i + 1
15.               B[lastc] ← B[lastc] | 0^{L−i−1}10^i
16.           End of if
17.       End of for
18.       I ← 0^L, F ← 0^L /* build I and F */
19.       For i ∈ 0...L − 1 Do
20.           If A & 0^{L−i−1}10^i ≠ 0^L Then
21.               If F & 0^{L−i}10^{i−1} = 0^L Then
22.                   I ← I | 0^{L−i}10^{i−1}
23.                   F ← F | 0^{L−i−1}10^i
24.               Else
25.                   F ← F & 1^{L−i}01^{i−1}
26.                   F ← F | 0^{L−i−1}10^i
27.               End of if
28.           End of if
29.       End of for
30.   Searching
31.       D ← 0^L
32.       For pos ∈ 1...n Do
33.           D ← ((D << 1) | 0^{L−1}1) & B[t_pos]) | (D & S[t_pos])
34.           Df ← D | F
35.           D ← D | (A & ((∼ (Df − I)) ∧ Df))
36.           If D & 10^{L−1} ≠ 0^L Then report an occurrence ending at pos
37.       End of for
```

Fig. 4.9. The extension of **Shift-And** to handle patterns with optional and repeatable characters.

1. Reading `a`

| | |
|---|---|
| $B$ | 0 0 0 0 0 1 |
| $S$ | 0 0 0 0 0 0 |
| $D$ | 0 0 0 0 0 1 |
| $Df$ | 0 0 0 1 0 1 |
| $D$ | 0 0 0 1 1 1 |

The propagation over the two optional characters `"b?c*"` took effect.

2. Reading `c`

| | |
|---|---|
| $B$ | 0 0 0 1 0 0 |
| $S$ | 0 0 0 1 0 0 |
| $D$ | 0 0 0 1 0 0 |
| $Df$ | 0 0 0 1 0 0 |
| $D$ | 0 0 0 1 0 0 |

This time there were no special propagation effects.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

3. Reading `c`

| | |
|---|---|
| $B$ | 0 0 0 1 0 0 |
| $S$ | 0 0 0 1 0 0 |
| $D$ | 0 0 0 1 0 0 |
| $Df$ | 0 0 0 1 0 0 |
| $D$ | 0 0 0 1 0 0 |

The $S$ table permitted the third bit of $D$ to stay active.

4. Reading `c`

| | |
|---|---|
| $B$ | 0 0 0 1 0 0 |
| $S$ | 0 0 0 1 0 0 |
| $D$ | 0 0 0 1 0 0 |
| $Df$ | 0 0 0 1 0 0 |
| $D$ | 0 0 0 1 0 0 |

5. Reading `d`

| | |
|---|---|
| $B$ | 0 0 1 0 0 0 |
| $S$ | 0 0 0 0 0 0 |
| $D$ | 0 0 1 0 0 0 |
| $Df$ | 0 0 1 1 0 0 |
| $D$ | 0 0 1 0 0 0 |

6. Reading `f`

| | |
|---|---|
| $B$ | 1 0 0 0 0 0 |
| $S$ | 0 0 0 0 0 0 |
| $D$ | 0 0 0 0 0 0 |
| $Df$ | 0 0 0 1 0 0 |
| $D$ | 0 0 0 0 0 0 |

7. Reading `a`

| | |
|---|---|
| $B$ | 0 0 0 0 0 1 |
| $S$ | 0 0 0 0 0 0 |
| $D$ | 0 0 0 0 0 1 |
| $Df$ | 0 0 0 1 0 1 |
| $D$ | 0 0 0 1 1 1 |

The propagation over the two optional characters `"b?c*"` took effect again.

8. Reading `b`

| | |
|---|---|
| $B$ | 0 0 0 0 1 0 |
| $S$ | 0 0 0 0 0 0 |
| $D$ | 0 0 0 0 1 0 |
| $Df$ | 0 0 0 1 1 0 |
| $D$ | 0 0 0 1 1 0 |

The propagation over the optional character `"c*"` took effect.

9. Reading `d`

| | |
|---|---|
| $B$ | 0 0 1 0 0 0 |
| $S$ | 0 0 0 0 0 0 |
| $D$ | 0 0 1 0 0 0 |
| $Df$ | 0 0 1 1 0 0 |
| $D$ | 0 0 1 0 0 0 |

No propagation effects this time. The previous propagation has allowed us to ignore a nonexistent `"c"` in the text.

10. Reading `e`

| | |
|---|---|
| $B$ | 0 1 0 0 0 0 |
| $S$ | 0 1 0 0 0 0 |
| $D$ | 0 1 0 0 0 0 |
| $Df$ | 0 1 0 1 0 0 |
| $D$ | 0 1 0 0 0 0 |

11. Reading `e`

| | |
|---|---|
| $B$ | 0 1 0 0 0 0 |
| $S$ | 0 1 0 0 0 0 |
| $D$ | 0 1 0 0 0 0 |
| $Df$ | 0 1 0 1 0 0 |
| $D$ | 0 1 0 0 0 0 |

The $S$ table permits the automaton to stay alive while it keeps reading `"e"`.

12. Reading `e`

| | |
|---|---|
| $B$ | 0 1 0 0 0 0 |
| $S$ | 0 1 0 0 0 0 |
| $D$ | 0 1 0 0 0 0 |
| $Df$ | 0 1 0 1 0 0 |
| $D$ | 0 1 0 0 0 0 |

13. Reading `f`

| | |
|---|---|
| $B$ | 1 0 0 0 0 0 |
| $S$ | 0 0 0 0 0 0 |
| $D$ | 1 0 0 0 0 0 |
| $Df$ | 1 0 0 1 0 0 |
| $D$ | 1 0 0 0 0 0 |

The last bit of $D$ is active, so we report an occurrence ending at text position 13.

### *4.5.2 Extended BNDM*

Figure 4.10 shows pseudo-code for the **BNDM** extension. The preprocessing for **Extended-BNDM** is the same except that the bits in the mask are in reverse order and we also compute $\ell min$. Note that the computation of $I$ and $F$ is unaltered even when our pattern is reversed, because the arithmetic operations always work in the same direction.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

```
Extended-BNDM (p = p_1 p_2 ... p_m, T = t_1 t_2 ... t_n)
1.    Preprocessing
2.        L ← number of normal characters in p
3.        ℓmin ← minimum length of an occurrence
4.        A ← 0^L /* build B, S and A */
5.        For c ∈ Σ Do B[c] ← 0^L, S[c] =← 0^L
6.        i ← −1
7.        For j ∈ 1...m Do
8.            If p_j = "+" Then S[lastc] ← S[lastc] | 0^i 10^{L−i−1}
9.            Else If p_j = "?" Then A ← A | 0^i 10^{L−i−1}
10.           Else If p_j = "*" Then
11.               S[lastc] ← S[lastc] | 0^i 10^{L−i−1}
12.               A ← A | 0^i 10^{L−i−1}
13.           Else /* p_j is a character */
14.               lastc ← p_j
15.               i ← i + 1
16.               B[lastc] ← B[lastc] | 0^i 10^{L−i−1}
17.           End of if
18.       End of for
19.       I ← 0^L, F ← 0^L /* build I and F */
20.       For i ∈ 0...L − 1 Do
21.           If A & 0^{L−i−1} 10^i ≠ 0^L Then
22.               If F & 0^{L−i} 10^{i−1} = 0^L Then
23.                   I ← I | 0^{L−i} 10^{i−1}
24.                   F ← F | 0^{L−i−1} 10^i
25.               Else
26.                   F ← F & 1^{L−i} 01^{i−1}
27.                   F ← F | 0^{L−i−1} 10^i
28.               End of if
29.           End of if
30.       End of for
31.   Searching
32.       pos ← 0
33.       While pos ≤ n − ℓmin Do
34.           j ← ℓmin − 1, last ← ℓmin
35.           D ← B[t_{pos+ℓmin}]
36.           If D & 10^{L−1} ≠ 0^L Then last ← j
37.           While D ≠ 0^L AND j > 0 Do
38.               Df ← D | F
39.               D ← D | (A & ((∼ (Df − I)) ∧ Df))
40.               D ← ((D << 1) & B[t_{pos+j}]) | (D & S[t_{pos+j}])
41.               j ← j − 1
42.               If D & 10^{L−1} ≠ 0^L Then /* prefix recognized */
43.                   If j > 0 Then last ← j
44.                   Else check a possible occurrence starting at pos
45.               End of if
46.           End of while
47.           pos ← pos + last
48.       End of while
```

Fig. 4.10. The extension of **BNDM** to handle patterns with optional and repeatable characters. It assumes $\ell min > 1$.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

The search is more complicated. We initialize $D$ using the last character of the window. Then the loop checks for a match and afterward processes the next window character. As for patterns with gaps, we need a forward verification for windows that may match the pattern.

The fact that the maximum length of an occurrence is in general unbounded for extended patterns makes it impossible to know beforehand what the maximum number of characters read will be when checking the occurrence of a pattern in the text window. We have to continue until the automaton runs out of active states, we find the pattern, or the text ends.

**Example of Extended-BNDM** We search for the initial position of the occurrences of the pattern `"ab?c*de+f"` in the text `"acccdfabdeeef"`.

| $c$ | $B$ | $S$ |
|---|---|---|
| a | 1 0 0 0 0 0 | 0 0 0 0 0 0 |
| b | 0 1 0 0 0 0 | 0 0 0 0 0 0 |
| c | 0 0 1 0 0 0 | 0 0 1 0 0 0 |
| d | 0 0 0 1 0 0 | 0 0 0 0 0 0 |
| e | 0 0 0 0 1 0 | 0 0 0 0 1 0 |
| f | 0 0 0 0 0 1 | 0 0 0 0 0 0 |
| * | 0 0 0 0 0 0 | 0 0 0 0 0 0 |

$$\begin{aligned} m &= 9 \\ L &= 6 \\ \ell min &= 4 \end{aligned}$$

$$\begin{aligned} A &= 0\,1\,1\,0\,0\,0 \\ I &= 0\,0\,0\,1\,0\,0 \\ F &= 0\,1\,0\,0\,0\,0 \end{aligned}$$

1. `accc` `dfabdeeef`

   $last \leftarrow 4$

| Reading `c` | $B$ | 0 0 1 0 0 0 |
|---|---|---|
| | $D$ | 0 0 1 0 0 0 |
| Reading `c` | $B$ | 0 0 1 0 0 0 |
| | $S$ | 0 0 1 0 0 0 |
| | $Df$ | 0 1 1 0 0 0 |
| | $D$ | 0 1 1 0 0 0 |
| | $D$ | 0 0 1 0 0 0 |
| Reading `c` | $B$ | 0 0 1 0 0 0 |
| | $S$ | 0 0 1 0 0 0 |
| | $Df$ | 0 1 1 0 0 0 |
| | $D$ | 0 1 1 0 0 0 |
| | $D$ | 0 0 1 0 0 0 |
| Reading `a` | $B$ | 1 0 0 0 0 0 |
| | $S$ | 0 0 0 0 0 0 |
| | $Df$ | 0 1 1 0 0 0 |
| | $D$ | 0 1 1 0 0 0 |
| | $D$ | 1 0 0 0 0 0 |

   The last bit of $D$ is set and $j = 0$, so we check forward the pattern in the text window `"acccdfa..."`. At the sixth character the automaton runs out of active states without finding the pattern. So we shift the window by $last = 4$.

2. `accc` `dfab` `deeef`

   $last \leftarrow 4$

| Reading `b` | $B$ | 0 1 0 0 0 0 |
|---|---|---|
| | $D$ | 0 1 0 0 0 0 |
| Reading `a` | $B$ | 1 0 0 0 0 0 |
| | $S$ | 0 0 0 0 0 0 |
| | $Df$ | 0 1 0 0 0 0 |
| | $D$ | 0 1 0 0 0 0 |
| | $D$ | 1 0 0 0 0 0 |

   The last bit of $D$ is set and $j > 0$, so we set $last \leftarrow 2$.

| Reading `f` | $B$ | 0 0 0 0 0 1 |
|---|---|---|
| | $S$ | 0 0 0 0 0 0 |
| | $Df$ | 1 1 0 0 0 0 |
| | $D$ | 1 0 0 0 0 0 |
| | $D$ | 0 0 0 0 0 0 |

   There are no more active states in $D$, so we shift by $last = 2$.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

3. `acccdf` `abde` `eef`

*last* ← 4

| | | |
|---|---|---|
| Reading `e` | *B* | 0 0 0 0 1 0 |
| | *D* | 0 0 0 0 1 0 |
| Reading `d` | *B* | 0 0 0 1 0 0 |
| | *S* | 0 0 0 0 0 0 |
| | *Df* | 0 1 0 0 1 0 |
| | *D* | 0 0 0 0 1 0 |
| | *D* | 0 0 0 1 0 0 |
| Reading `b` | *B* | 0 1 0 0 0 0 |
| | *S* | 0 0 0 0 0 0 |
| | *Df* | 0 1 0 1 0 0 |
| | *D* | 0 1 1 1 0 0 |
| | *D* | 0 1 0 0 0 0 |
| Reading `a` | *B* | 1 0 0 0 0 0 |
| | *S* | 0 0 0 0 0 0 |
| | *Df* | 0 1 0 0 0 0 |
| | *D* | 0 1 0 0 0 0 |
| | *D* | 1 0 0 0 0 0 |

The last bit of $D$ is active and $j = 0$, so we perform a forward check on the text window `"abdeeef"`. We find an occurrence, so we report the seventh text position as the beginning of an occurrence. Then we shift the window by $last = 4$.

This puts the window outside the text, so we are finished.

## 4.6 Multipattern searching

Consider now the problem of searching a number of extended strings simultaneously. Since the only techniques that deal well with extended strings are based on bit-parallelism, we need a multipattern search algorithm based on bit-parallelism. Unfortunately, as seen in Chapter 3, most of the techniques for multipattern search do not use bit-parallelism.

The only approach useful for us is the one considered in Sections 3.2.1 and 3.4.1, which packs a number of automata into a single computer word and performs **Shift-And**– or **BNDM**–like searching. If we are searching a number of extended strings of the same kind, we can use the same technique: We pack the bits of many automata in a single computer word and simulate the corresponding type of search on the whole word, thus updating the states of the automata represented in there. As for simple strings, we need to take care of the limits between different patterns and of the initial self-loops of the automata.

This multipattern search capability is extremely limited, as we will be able to represent just a few extended patterns in a single computer word.

When trying to extend **BNDM** in Section 3.4.1 we assumed that all the strings had the same length and otherwise truncated them to the shortest one. Here we do analogously: The $\ell min$ values of the patterns may be different, and we truncate them to obtain patterns with the same $\ell min$ value.

The truncation in Section 3.4.1 requires checking forward in the window for the presence of the complete pattern. This does not involve extra complications here, because we *need* to perform a forward verification with the whole patterns that seem to occur in the window.

https://doi.org/10.1017/CBO9781316135228.004 Published online by Cambridge University Press

The easiest way to do the truncation is to take the longest possible pattern prefix whose $\ell min$ is as chosen, although it is possible to take a pattern factor that has a lower probability of matching. This optimization is pursued in [Nav01b]. Note that the verification is more complex in this case because we have to verify in front of and behind the window in the text.

## 4.7 Other algorithms and references

The problem of string matching with "don't cares" is a simplification of what we have presented under the name "classes of characters." In this problem there are pattern and text positions whose value is the whole class $\Sigma$. An algorithm with time complexity $O(n \log^2 n)$ exists for this problem [FP74]. It is based on convolutions.

The same paper [FP74] presents an $O(n \log^2 m \log\log m \log |\Sigma|)$ time algorithm for patterns with wild cards. For the same problem, an $O(n + m\sqrt{n} \log n \sqrt{\log\log n})$ time algorithm is presented in [Abr87]. The work [Pin85] obtains the same complexity as [FP74] for classes of characters where complements of single characters are permitted. The work [Abr87] considers general classes of characters and obtains subquadratic search algorithms.

All these algorithms are theoretically interesting but are hardly usable in practice. A good survey on the open theoretical problems and existing results in nonstandard stringology is [MP94].

Extensions to patterns with gaps are of great interest in computational biology. For example, one may permit gaps of negative lengths, where some parts of the pattern appear superimposed in the text. These patterns are considered in [MM89, KM95, Mye96], where they also are searched approximately. They are covered in more detail in Chapter 6.