3

Multiple string matching

3.1 Basic concepts

The single string matching problem may be extended in a natural way to search simultaneously for a set of strings $P = \{p^1, p^2, \ldots p^r\}$, where each p^i is a string $p^i = p^i_1 p^i_2 \ldots p^i_{m_i}$ over a finite character set Σ. Denote by $|P|$ the sum of the lengths of the strings in P, more formally $|P| = \sum_{i=1}^{r} |p^i| = \sum_{i=1}^{r} m_i$. Let ℓmin be the minimum length of a pattern in P and ℓmax the maximum. As before, the search is done in a text $T = t_1 t_2 \ldots t_n$.

Strings in P may be factors, prefixes, suffixes, or even the same as others. For example, if we search for the set $\{\texttt{ATATA}, \texttt{TATA}\}$ in a DNA sequence, each time we find an occurrence of `ATATA` we also find an occurrence of the second string. Hence, the total number of occurrences can be $r \times n$. To make the multistring matching problem precise, we consider that we are interested in reporting all pairs (i, j) such that $t_{j-|p^i|+1} \ldots t_j$ is equal to p^i.

The simplest solution to this problem is to repeat r searches with one of the algorithms of Chapter 2. This leads to a total worst-case complexity of $O(|P|)$ for the preprocessing and $O(r \times n)$ for the search.

The worst-case search complexity can be reduced to $O(n + nocc)$, where $nocc$ is the total number of occurrences, by using some kind of *extension* of the search algorithms for a single pattern. The average complexity can also be improved, although it is difficult to think in terms of "average" complexity, since many parameters play a role in the running time of the algorithms. The most important parameters are the size of the alphabet, the number of patterns, the distribution of the lengths of the patterns (particularly the minimum size), and the memory available.

We again denote by θ an object that is not defined. For instance, when we write **While** $q \neq \theta$ **Do**, it means we iterate while q is defined.

Troughout this chapter, we will consider the example in Figure 3.1. We

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

will simultaneously search for the three strings "`announce`", "`annual`", and "`annually`".

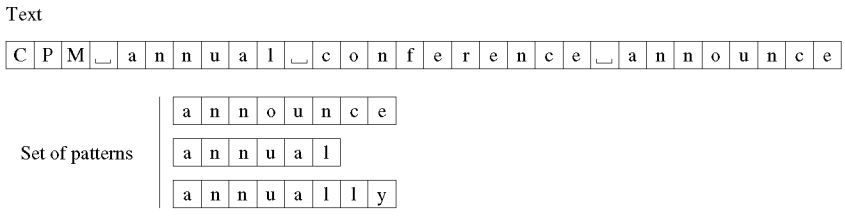

Fig. 3.1. Simultaneously searching three strings in our example text.

As with a single string, sets of natural language strings usually contain fewer repetitions than sets of DNA sequences. To show the tricky cases that could occur, we also show the behavior of our algorithms when searching for the set of strings `ATATATA,TATAT,ACGATAT` in the sequence `AGATACGATATATAC`.

The three approaches for searching a single string (Chapter 2) lead to several extensions for searching a set of strings. For each approach, there are usually many possible extensions, according to the way the set of patterns is managed and the way the shifts are obtained. The notion of a *search window* is not relevant for multiple string matching, which will become clear soon. We present in this chapter the empirically most efficient extensions, which are usually also the simplest.

Prefix searching (Figure 3.2) The search is done forward, reading the characters of the text one after another with an automaton built on the set P. For each position of the text, we compute through this automaton the longest suffix of the text read that is also a prefix of one of the strings of P. The most famous algorithm that uses this approach is **Aho-Corasick** [AC75].

Suffix searching (Figure 3.3) A position *pos* is slid along the text, from which we search backward for a suffix of any of the strings. As with a single pattern, we shift *pos* according to the next occurrence of the suffix read in P. This approach may avoid reading all the characters of the text.

Factor searching (Figure 3.4) A position *pos* is also slid along the text, from which we read backwards a factor of some prefix of size ℓmin of the strings in P. It also may avoid reading all the characters of the text.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

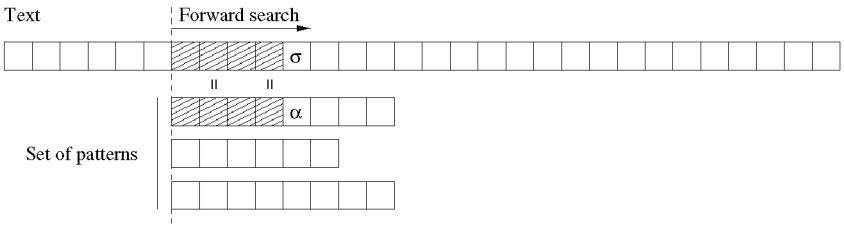

Fig. 3.2. First approach: We compute the longest prefix of a pattern in the set that is also a suffix of the text read. It requires reading all the characters of the text at least once.

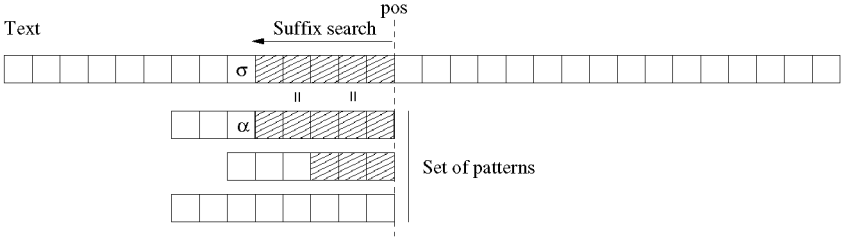

Fig. 3.3. Second approach: We search backwards for a suffix of one of the strings. It avoids, on average, reading all characters of the text.

Before describing these three approaches in depth, we introduce a basic data structure on a set of strings, called a *trie*. This structure is used by most of the classical multistring matching algorithms. The *trie* of the set $P = \{p^1, p^2, \ldots p^r\}$ is a rooted directed tree that represents the set P; that is, every path starting from the root is labeled by one of the strings p^i, and, conversely, every string $p^i \in P$ labels a path from the root. Below, unless specified, paths start at the root. Every state q corresponding to an entire string is marked as *terminal*, and a function $F(q)$ points to a list of

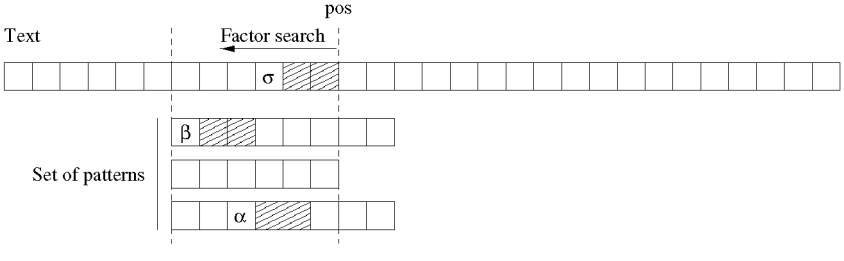

Fig. 3.4. Third approach: We search for a factor of any of the patterns in the current window.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

all the numbers of the strings in P that correspond to q. We give the trie for $P = \{$`announce`,`annual`,`annually`$\}$ in Figure 3.5.

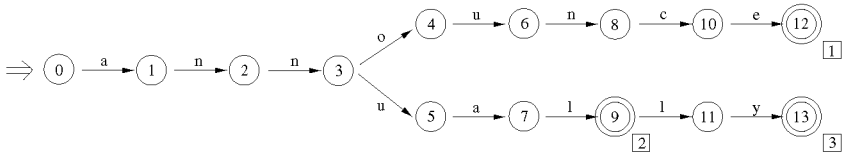

Fig. 3.5. Trie for $P =$ {`announce`, `annual`, `annually`}. The function F of each terminal state is represented with squares. It indicates the identifier of the string in P.

We usually use the automata notation for describing a trie, since a trie is also a deterministic acyclic automaton recognizing the corresponding set of strings. The trie of a set P can be built in $O(|P|)$ time by inserting the strings p^i one by one into the tree, starting at the root, and building the corresponding transitions. Pseudo-code for the trie construction is given in Figure 3.6.

Trie($P = \{p^1, p^2, \ldots, p^r\}$)
1. Create an initial non terminal state 0
2. **For** $i \in 1 \ldots r$ **Do**
3. $\quad$ *Current* $\leftarrow$ initial state 0
4. $\quad$ $j \leftarrow 1$
5. $\quad$ **While** $j \leq m_i$ AND $\delta(Current, p^i_j) \neq \theta$ **Do**
6. $\quad\quad$ *Current* $\leftarrow \delta(Current, p^i_j)$
7. $\quad\quad$ $j \leftarrow j + 1$
8. $\quad$ **End of while**
9. $\quad$ **While** $j \leq m_i$ **Do**
10. $\quad\quad$ Create a new non terminal state *State*
11. $\quad\quad$ $\delta(Current, p^i_j) \leftarrow State$
12. $\quad\quad$ *Current* $\leftarrow$ *State*
13. $\quad\quad$ $j \leftarrow j + 1$
14. $\quad$ **End of while**
15. $\quad$ **If** *Current* is already terminal **Then** $F(Current) \leftarrow F(Current) \cup \{i\}$
16. $\quad$ **Else** mark *Current* as terminal, $F(Current) \leftarrow \{i\}$
17. **End of for**

Fig. 3.6. Pseudo-code for the construction of a trie from a set of strings $P = \{p^1, p^2, \ldots, p^r\}$. The strings are taken one by one and inserted into the tree.

The size of the trie depends on the implementation of the transitions. The simplest implementation is for each state q of the trie to code $\delta(q, *)$ in a table of size $|\Sigma|$. Then the total size of the trie of a set P is worst-case $|\Sigma| \times |P|$. This representation is usually used when the sizes of the set

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

of strings and of the alphabet are not too large. It has the advantage of passing through a transition in constant time $O(1)$ by performing an access to a table.

Since the total number of transitions is at most $|P|$, it is possible to code all the transitions in $O(|P|)$ space, independently of the size of the alphabet. However, the time to pass through a transition increases. If the transitions of each state are coded with a linked list, sorted or not, this time grows to $O(|\Sigma|)$ in the worst and average cases. It can be reduced to $O(\log |\Sigma|)$ by coding the transitions with balanced trees [CLR90], but this complicates the code.

We now describe in detail the three general approaches to search for a set of strings.

3.2 Prefix based approach

The extension of the prefix based approach leads to the **Multiple Shift-And** and **Aho-Corasick** algorithms. As with a single pattern (Section 2.2), we assume that we have read the text up to position i and that we know the length of the longest suffix of $t_1 \dots t_i$ that is a prefix of a pattern $p^k \in P$. The algorithmic problem is to calculate this length after reading the next character of the text.

In the single pattern case, there were two ways of finding this length. One was based on managing a bit array with bit-parallelism. For multiple pattern matching, this technique is only practical for very small patterns, because the total length $|P|$ has to be smaller than a few computer words. Nevertheless, this possibility is widely used for extended string matching (Chapter 4) and approximate string matching (Chapter 6). We call this algorithm **Multiple Shift-And**.

The solution, when the length of the set is too large to fit in computer words, is to find a mechanism that computes the size of the longest suffix of the text read that is also a prefix of one of the strings of P, in amortized constant time per character. This is what the **Aho-Corasick** algorithm does, with a linear time $O(|P|)$ preprocessing phase.

3.2.1 Multiple Shift-And algorithm

The bit-parallelism approach is only valuable when the set $P = \{p^1, \dots, p^r\}$ has a total length $|P|$ small enough to fit in a few computer words. For simplicity, we assume below that $|P|$ is smaller than w. The idea is to perform with bit-parallelism all the computations required by the **Shift-**

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

And algorithm (Section 2.2.2) for the r strings in the same computer word [BYG89b].

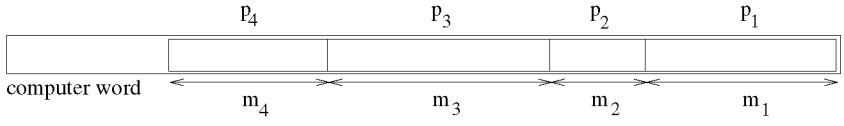

Fig. 3.7. **Multiple Shift-And** algorithm. The total size of the patterns has to fit in w.

We pack the strings together in the computer word, as in Figure 3.7. Then, for each new character of the text, we perform the computations for the strings of P like in the **Shift-And** algorithm. The initialization word DI is the concatenation of the initialization words for each string, that is,

$$DI \leftarrow 0^{m_r-1}1 \ldots 0^{m_2-1}10^{m_1-1}1$$

Similarly, the final test is

$$DF \leftarrow 10^{m_r-1} \ldots 10^{m_2-1}10^{m_1-1}$$

The main loop is the same as for the **Shift-And** algorithm. Pseudo-code is given in Figure 3.8.

The **Shift-Or** trick (Section 2.2.2) cannot be used here, since the shift "<<" only introduces a zero to the right, and we need a zero in each position that begins a new string of P in the computer word.

Example using English We search for the set of strings $P = \{$`announce, annual, annually`$\}$ in the text "`annual_announce`".

Table B

a	0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1
c	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
e	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
l	0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
n	0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 1 0 0 1 1 0
o	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
u	0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
y	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

$DI = 0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1$
$DF = 1\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0$
$D \;\; = 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0$

1. Reading `a`

$$\begin{array}{ll} & 0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1 \\ B[\texttt{a}] & 0\,0\,0\,1\,0\,0\,0\,1\,0\,1\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1 \end{array}$$

2. Reading `n`

$$\begin{array}{ll} & 0\,0\,0\,0\,0\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0\,0\,0\,0\,0\,0\,1\,1 \\ B[\texttt{n}] & 0\,0\,0\,0\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0\,0\,0\,1\,0\,0\,1\,1\,0 \\ \hline D = & 0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1\,1\,0 \end{array}$$

3. Reading `n`

$$\begin{array}{ll} & 0\,0\,0\,0\,0\,1\,0\,1\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0\,1\,0\,1 \\ B[\texttt{n}] & 0\,0\,0\,0\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0\,0\,0\,1\,0\,0\,1\,1\,0 \\ \hline D = & 0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0 \end{array}$$

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

Multiple Shift-And$(P = \{p^1, p^2, \ldots, p^r\}, T = t_1 t_2 \ldots t_n)$

```
1.  Preprocessing
2.      For c ∈ Σ Do B[c] ← 0^|P|
3.      ℓ ← 0
4.      For k ∈ 1...r Do
5.          For j ∈ 1...m_k Do B[p_j^k] ← B[p_j^k] | 0^(|P|−ℓ−j) 10^(ℓ+j−1)
6.          ℓ ← ℓ + m_k
7.      End of for
8.      DI ← 0^(m_r−1) 1 ... 0^(m_2−1) 10^(m_1−1) 1
9.      DF ← 10^(m_r−1) ... 10^(m_2−1) 10^(m_1−1)
10. Searching
11.     D ← 0^|P|
12.     For pos ∈ 1...n Do
13.         D ← ((D << 1) | DI)  &  B[t_pos]
14.         If D & DF ≠ 0^|P| Then
15.             Check which patterns match
16.             Report the corresponding occurrences ending in pos
17.         End of if
18.     End of for
```

Fig. 3.8. **Multiple Shift-And** algorithm. The total length of the patterns $|P|$ has to be less than w. We let $m_k = |p^k|$.

4. Reading `u`

	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1											
B[u]	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0												
$D =$	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

5. Reading `a`

	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1											
B[a]	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1											
$D =$	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

6. Reading `l`

	0	0	1	0	0	0	1	1	1	0	0	0	1	1	0	0	0	0	0	0	1	1											
B[l]	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0											
$D =$	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

D & $DF \neq 0^{|P|}$, we check the patterns that match, and we mark an occurrence of `annual`.

7. Reading `_`

	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1											
B[_]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0											
$D =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

8. Reading `a`

	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1											
B[a]	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	1											
$D =$	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

9. Reading `n`

	0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	1	1										
B[n]	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	1	0	0	1	1	0											
$D =$	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

10. Reading `n`

	0	0	0	0	0	1	0	1	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1										
B[n]	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	1	0	0	1	1	0											
$D =$	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

11. Reading `o`

	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	1											
B[o]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0										
$D =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

12. Reading `u`

	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1											
B[u]	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0												
$D =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

13. Reading `n`

	0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 1
B[`n`]	0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 1 0 0 1 1 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

14. Reading `c`

	0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1
B[`c`]	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

15. Reading `e`

	0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1
B[`c`]	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$, we check the patterns that match, and we mark an occurrence of `announce`.

Example using DNA We search for the set of strings P = {`ATATATA`, `TATAT`, `ACGATAT`} in the text `AGATACGATATATAC`.

Table B

`A`	0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1
`C`	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
`G`	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
`T`	1 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0
`*`	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

.

$DI =$	0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1
$DF =$	1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1. Reading `A`

	0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1
B[`A`]	0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1
$D =$	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1

2. Reading `G`

	0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 1 1
B[`G`]	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

3. Reading `A`

	0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1
B[`A`]	0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1
$D =$	0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1

4. Reading `T`

	0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 1 1
B[`T`]	1 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0

5. Reading `A`

	0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 1
B[`A`]	0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1
$D =$	0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1

6. Reading `C`

	0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 1 0 1 1
B[`C`]	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
$D =$	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

7. Reading `G`

	0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 1
B[`G`]	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$D =$	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

8. Reading `A`

	0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1
B[`A`]	0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1
$D =$	0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1

9. Reading `T`

	0 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 1 1
B[`T`]	1 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0
$D =$	0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0

10. Reading `A`

	0 1 0 0 0 0 1 0 0 0 1 1 0 0 0 0 1 0 1
B[`A`]	0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1
$D =$	0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1

11. Reading `T`

	1 0 0 0 0 1 1 0 0 1 0 1 0 0 0 1 0 1 1
B[`T`]	1 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0
$D =$	1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0

D & $DF \neq 0^{|P|}$, we check the patterns that match, and we mark an occurrence of `ACGATAT`.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

12. Reading A

	0	0	0	0	0	0	1	0	1	0	1	1	0	0	1	0	1	0	1
B[A]	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$D =$	0	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	1

13. Reading T

	0	0	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	1	1
B[T]	1	0	1	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0
$D =$	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0

D & $DF \neq 0^{|P|}$, we check the patterns that match, and we mark an occurrence of ATATAT.

14. Reading A

	0	0	0	0	0	0	1	0	1	0	1	1	1	0	1	0	1	0	1
B[A]	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1
$D =$	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0

D & $DF \neq 0^{|P|}$, we check the patterns that match, and we mark an occurrence of ATATATA.

15. Reading C

	0	0	0	0	0	1	1	1	0	1	0	1	0	1	0	1	0	1	1
B[C]	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$D =$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3.2.2 Basic Aho-Corasick algorithm

The algorithm of Aho and Corasick [AC75] is an extension of the **Knuth-Morris-Pratt** algorithm (Section 2.2.1) for a set of patterns.

The algorithm uses a special automaton, called the *Aho-Corasick automaton*, built on P. It is the trie of P augmented with a "supply function" S_{AC}.

Formally, we denote by q a state of the trie of P, and by $L(q)$ the label of the path from the initial state to q. Then $S_{AC}(q)$ is defined, except for the initial state, as the state reached when the automaton reads the longest suffix of $L(q)$ that is also a prefix of some $p^i \in P$. This is a kind of extension of a *border* (Section 2.2.1) to a set of strings. The supply state of the initial state is set to θ. A *supply link* goes from each state q to $S_{AC}(q)$, and a *supply path* is a chain of supply links.

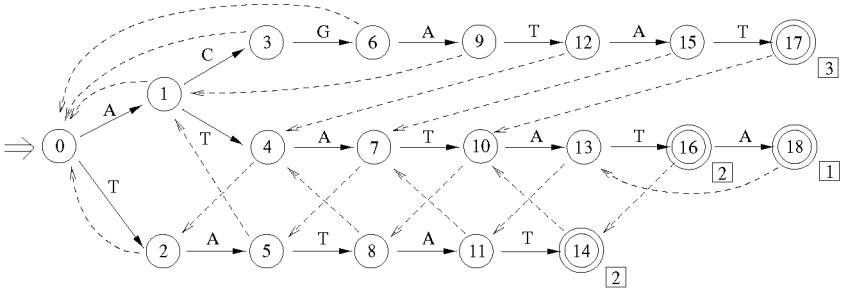

Fig. 3.9. Aho-Corasick automaton for the set {ATATATA, TATAT, ACGATAT}. The dashed links represent the state-to-state supply function S_{AC}. Double-circled states are terminal.

The Aho-Corasick automaton for the set {ATATATA, TATAT, ACGATAT} is shown in Figure 3.9. On this automaton, for instance, $L(15) =$ ACGATA, its longest suffix that is also a prefix of one of the patterns is ATA, which leads

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

to state 7, and hence $S_{AC}(15) = 7$. The terminal states are those of the trie that correspond to an entire pattern, and also all states whose supply paths go down through another terminal state on their way to the root. In Figure 3.9, for instance, state 16 is terminal because $S_{AC}(16)$ is terminal.

We assume that a prefix $t_1 t_2 \dots t_i$ of the text has already been read, and that the longest suffix of $t_1 \dots t_i$ that is also a prefix of one of the patterns leads to a state *Current* in the Aho-Corasick automaton. We denote this longest suffix $v = L(Current)$. We want to read t_{i+1} and compute for $t_1 \dots t_i t_{i+1}$ the new longest suffix u. There are two cases.

1. If there exists an outgoing transition from *Current* to another state f in the trie labeled by t_{i+1}, then the new *Current* state becomes f, and $u = L(f) = ut_{i+1}$ is the new longest prefix of one of the patterns that is a suffix of $t_1 \dots t_{i+1}$.
2. If not (i.e., , we fail reading t_{i+1} in the tree), we go down the supply path of q until either
   (a) we find a state on the path followed by t_{i+1}. In this case, the current state becomes the arrival state f by the transition t_{i+1}, and $u = L(f)$.
   (b) we reach θ, which means that the longest suffix u we search for is the empty string ε, and we move to the initial state.

Pseudo-code for the search algorithm is given in Figure 3.10. The complexity of the search phase is simple to evaluate, if we observe that we cannot go down more supply links than text characters we read. The number of supply links crossed through is then bounded by n, and the number of transitions used (real transitions plus supply links) is bounded by $2n$. The number of character comparisons depends on how the transitions of the automaton are implemented. The complexity is $O(n + nocc)$ if they are coded with a table, and $O(n \log |\Sigma| + nocc)$ with balanced trees.

To construct the Aho-Corasick automaton we begin by building the trie of the set of strings P with the algorithm in Figure 3.6. The states of the Aho-Corasick automaton are those of the trie. The initial state is the same and the terminal states of the trie are also terminal. We build the supply function S_{AC} on this trie in transversal order, which is the order we numbered the states in Figure 3.9.

We assume that we have computed the supply function of all the states before state *Current* in transversal order. We consider the parent *Parent* of *Current* in the trie, leading to *Current* by σ, that is, $Current = \delta_{AC}(Parent, \sigma)$. The supply state $S_{AC}(Parent)$ has already been computed. We search

```
Aho-Corasick(P = {p^1, p^2, ..., p^r}, T = t_1 t_2 ... t_n)
1.    Preprocessing
2.        AC ← Build_AC(P)
3.    Searching
4.        Current ← Initial state of the automaton AC
5.        For pos ∈ 1 ... n Do
6.            While δ_AC(Current, t_pos) = θ AND S_AC(Current) ≠ θ Do
7.                Current ← S_AC(Current)
8.            End of while
9.            If δ_AC(Current, t_pos) ≠ θ Then
10.               Current ← δ_AC(Current, t_pos)
11.           Else Current ← initial state of AC
12.           End of if
13.           If Current is terminal Then
14.               Mark all the occurrences (F(Current), pos)
15.           End of if
16.       End of for
```

Fig. 3.10. **Aho-Corasick** algorithm to search for a set $P = \{p^1, p^2, \ldots, p^r\}$ of strings. It uses the Aho-Corasick automaton to compute at each text character t_{pos} the longest prefix of any pattern p^k that is also a suffix of the text read $t_1 \ldots t_{pos}$.

for the state where u ends, u being the longest suffix of $v = L(Current)$ that labels a path in the trie. The string v has the form $v'\sigma$. If there exists such a nonempty string u, since it is a suffix of v, it must be of the form $u = u'\sigma$. In that case, u' is a suffix of v' that is the label of a path in the trie.

If $S_{AC}(Parent)$ has an outgoing transition by σ to a state h, then $w = L(S_{AC}(Parent))$ is the longest suffix of v' that is the label of a path, and $w\sigma$ is also a label of a path in the trie. Consequently, it is the longest suffix u of $v = v'\sigma$ that we are searching for, and $S_{AC}(Current)$ has to be set to h.

If $S_{AC}(Parent)$ does not have an outgoing transition by σ, we consider $S_{AC}(S_{AC}(Parent))$ and so on. We repeat the operation, until either we find a state on the supply path that has an outgoing transition by σ, or we find θ, which means that u is the empty string ε and $S_{AC}(Current)$ has to be set to the initial state.

The mechanism is similar to the **Aho-Corasick** search algorithm itself. Its pseudo-code is given in Figure 3.11. Complexity is evaluated with the observation we made for the whole algorithm: We do not go down more supply links than the total number of real transitions, which is bounded by $O(|P|)$. So the number of total transitions used (real transitions plus supply links) is bounded by $2 \times |P|$. Like for the search phase, the complexity in terms of comparisons of characters depends on how the transitions of the

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

automaton are implemented. It is $O(|P|)$ if they are coded with a table, and $O(|P| \log |\Sigma|)$ with balanced trees.

Build_AC($P = \{p^1, p^2, \ldots, p^r\}$)
1. $AC\_trie \leftarrow$ **Trie**(P)
   δ_{AC} is its transition function
2. $Initial\_state \leftarrow$ root of AC_trie
3. $S_{AC}(Initial\_state) \leftarrow \theta$
4. **For** *Current* in transversal order **Do**
5. $Parent \leftarrow$ parent of *Current* in AC_trie
6. $\sigma \leftarrow$ label of the transition from *Parent* to *Current*
7. $Down \leftarrow S_{AC}(Parent)$
8. **While** $Down \neq \theta$ AND $\delta_{AC}(Down, \sigma) = \theta$ **Do**
9. $Down \leftarrow S_{AC}(Down)$
10. **End of while**
11. **If** $Down \neq \theta$ **Then**
12. $S_{AC}(Current) \leftarrow \delta_{AC}(Down, \sigma)$
13. **If** $S_{AC}(Current)$ is terminal **Then**
14. Mark *Current* as terminal
15. $F(Current) \leftarrow F(Current) \cup F(S_{AC}(Current))$
16. **End of if**
17. **Else** $S_{AC}(Current) \leftarrow Initial\_state$
18. **End of if**
19. **End of for**

Fig. 3.11. Construction of the Aho-Corasick automaton. The state *Current* goes in transversal order through the trie AC_trie built on P. The state *Down* goes down the supply links from the parent of *Current*, looking for an outgoing transition labeled with the same character as between *Current* and its parent. $F(Current)$ is initialized as empty when *Current* is first marked as terminal.

Example using English We search for the set of strings $P =$ {`announce`, `annual`, `annually`} in the text "`annual_announce`". The Aho-Corasick automaton built on P is shown in Figure 3.12.

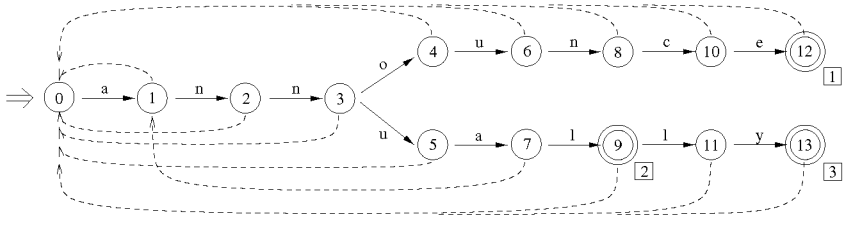

Fig. 3.12. Aho-Corasick automaton of our example set $P =$ {`announce`, `annual`, `annually`}. Double-circled states are terminal.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

Current $\leftarrow 0$.

1. Reading `a`
   Current $\leftarrow 1 = \delta(0, a)$
2. Reading `n`
   Current $\leftarrow 2 = \delta(1, n)$
3. Reading `n`
   Current $\leftarrow 3 = \delta(2, n)$
4. Reading `u`
   Current $\leftarrow 5 = \delta(3, u)$
5. Reading `a`
   Current $\leftarrow 7 = \delta(5, a)$
6. Reading `l`
   Current $\leftarrow 9 = \delta(7, a)$.
   The state 9 is terminal; we mark an occurrence of $F(9) \rightarrow$ `annual`.
7. Reading `_`
   $\delta(9, \_) = \theta$. We jump to $0 = S_{AC}(9)$.
   $\delta(0, \_) = \theta$, we jump to $\theta = S_{AC}(0)$.
   We continue the search from the initial state 0, *Current* $\leftarrow 0$.
8. Reading `a`
   Current $\leftarrow 1 = \delta(0, a)$
9. Reading `n`
   Current $\leftarrow 2 = \delta(1, n)$
10. Reading `n`
    Current $\leftarrow 3 = \delta(2, n)$
11. Reading `o`
    Current $\leftarrow 4 = \delta(3, o)$
12. Reading `u`
    Current $\leftarrow 6 = \delta(4, u)$
13. Reading `n`
    Current $\leftarrow 8 = \delta(6, n)$
14. Reading `c`
    Current $\leftarrow 10 = \delta(8, c)$
15. Reading `e`
    Current $\leftarrow 12 = \delta(10, e)$.
    The state 12 is terminal; we mark an occurrence of $F(12) \rightarrow$ `announce`.

Example using DNA We search for the set of strings $P = \{$`ATATATA, TATAT, ACGATAT`$\}$ in the text `AGATACGATATATAC`. We again use the Aho-Corasick automaton built on P already shown in Figure 3.9.

Current $\leftarrow 0$.

1. Reading `A`
   Current $\leftarrow 1 = \delta(0, A)$
2. Reading `G`
   $\delta(1, G) = \theta$. We jump to $0 = S_{AC}(1)$.
   $\delta(0, G) = \theta$; we jump to $\theta = S_{AC}(0)$.
   We continue the search from the initial state 0, *Current* $\leftarrow 0$.
3. Reading `A`
   Current $\leftarrow 1 = \delta(0, A)$
4. Reading `T`
   Current $\leftarrow 4 = \delta(1, T)$
5. Reading `A`
   Current $\leftarrow 7 = \delta(4, A)$
6. Reading `C`
   $\delta(7, C) = \theta$. We jump to $5 = S_{AC}(7)$.
   $\delta(5, C) = \theta$; we jump to $1 = S_{AC}(7)$.
   $\delta(1, C) = 3$, *Current* $\leftarrow 3$.
7. Reading `G`
   Current $\leftarrow 6 = \delta(3, G)$
8. Reading `A`
   Current $\leftarrow 9 = \delta(6, A)$

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

9. Reading `T`
   $Current \leftarrow 12 = \delta(9, T)$

10. Reading `A`
    $Current \leftarrow 15 = \delta(12, A)$

11. Reading `T`
    $Current \leftarrow 17 = \delta(12, T)$. The state 17 is terminal; we mark an occurrence of $F(17) \rightarrow$ `ACGATAT`.

12. Reading `A`
    $\delta(17, A) = \theta$. We jump to $10 = S_{AC}(17)$. $\delta(10, A) = 13$, $Current \leftarrow 13$.

13. Reading `T`
    $Current \leftarrow 16 = \delta(13, T)$. The state 16 is terminal; we mark an occurrence of $F(16) \rightarrow$ `TATAT`.

14. Reading `A`
    $Current \leftarrow 18 = \delta(16, A)$. The state 18 is terminal; we mark an occurrence of $F(18) \rightarrow$ `ATATATA`.

15. Reading `C`
    $\delta(18, C) = \theta$. We jump to $13 = S_{AC}(18)$. $\delta(13, C) = \theta$; we jump to $11 = S_{AC}(13)$. $\delta(11, C) = \theta$; we jump to $7 = S_{AC}(11)$. $\delta(7, C) = \theta$; we jump to $1 = S_{AC}(7)$. $\delta(1, C) = 3$, $Current \leftarrow 3$.

3.2.3 Advanced Aho-Corasick algorithm

The above algorithm permits a powerful variant. The idea is to precompute all the transitions simulated by the supply function. We then obtain a complete automaton (all the states have an outgoing transition by every character of the alphabet) that we name the *extended Aho-Corasick automaton.*

This completion can be computed using the supply function. We first complete the outgoing transitions of the initial state with a loop, which means $\delta(0, \sigma) \leftarrow 0$ for each new letter σ. Now, let *Current* be a state of the automaton taken in transversal order. We compute the missing outgoing transitions of *Current* by using the formula $\delta(Current, \sigma) = \delta(S_{AC}(Current), \sigma)$ for each new letter σ.

The drawback to this automaton is the large amount of memory space it requires. It is $O(|P| \times |\Sigma|)$ independently of the way the transitions are implemented. This construction is useful for relatively small sets and alphabets. A trade-off that is often used is to compute the new transitions on *the fly* if there is memory left. This was done in the first version of the well-known Unix application *Grep.*

3.3 Suffix based approach

The experimental results of Chapter 2 show that the suffix based approach is usually faster than the prefix based one. So it is natural to try to extend

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

the suffix based approach to sets of patterns. The first attempt was that of Commentz-Walter in 1979 [CW79]. It is a direct extension of the **Boyer-Moore** algorithm. The **Horspool** algorithm has also been extended, but it is much less powerful for multiple string matching than for single patterns. A stronger extension is the **Wu-Manber** algorithm, which is practical, simple, and efficient.

3.3.1 Commentz-Walter idea

The **Commentz-Walter** [CW79] algorithm is a "natural" extension of the **Boyer-Moore** algorithm (Section 2.3.1). This algorithm is never faster in practice than **Aho-Corasick** or other algorithms presented below. However, it is historically important because it was the first expected sublinear multistring matching algorithm, and it was implemented in the second version of the Unix application *Grep*. Currently, this algorithm does not have a real case of application, and we just present the idea it is based on.

The **Commentz-Walter** algorithm represents $P = \{p^1, \ldots, p^r\}$ using a trie of the reverse patterns $P^{rv} = \{(p^1)^{rv}, \ldots, (p^r)^{rv}\}$ inside which the text is read. A position *pos* is slid along the text, beginning at position ℓmin so as not to skip a possible occurrence. For each such new position, we read backwards the longest suffix u of $t_1 \ldots t_{pos}$ that is also a suffix of one of the patterns. If we find an occurrence, we mark it. Then, we shift the position of the search to the right, using the three functions d_1, d_2, d_3 of the **Boyer-Moore** algorithm extended to a set of strings. The first two functions are computed for each state of the trie, and to shift we consider them at the last state q we crossed when reading the longest suffix u.

- $d_1(q)$ is the minimal shift such that $u = L(q)$ matches a factor of some $p^j \in P$.
- $d_2(q)$ is the minimal shift such that a suffix of $u = L(q)$ matches a prefix of some $p^j \in P$.

The last function $d_3[\alpha, k]$ is computed for each character α of the alphabet for positions $0 \leq k < \ell max$. It is the minimal shift such that α read at position $pos - k$ matches another character of some $p^j \in P$.

For a visual idea of what these three functions do, the reader may refer to Figures 2.8, 2.9, and 2.10 of Chapter 2, where the three corresponding functions of the **Boyer-Moore** algorithm are shown.

We combine these three functions to compute a shift. Suppose that we read backwards k characters of the text from a position *pos* and this led to

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

state q. The shift $s[q, pos, k]$ is then computed with the following formula:

$$s[q, pos, k] = \min \begin{cases} \max(d_1[q], d_3[t_{pos-k}, k]) \\ d_2[q] \end{cases}$$

The formula is the direct extension of the computation of the window shifts in the **Boyer-Moore** algorithm. As $d_2 \leq \ell min$, the longest shift is bounded by ℓmin, which is a necessary condition to avoid skipping an occurrence when shifting the position pos.

The **Commentz-Walter** algorithm is worst-case time $O(n \times \ell max)$ but sublinear on average if the number of patterns is not too large. The computation of the three functions d_1, d_2, and d_3 can be done in $O(|P|)$ time.

3.3.2 Set Horspool algorithm

The **Horspool** algorithm, similarly to **Boyer-Moore**, is directly extensible to a set of patterns. The new algorithm, which we call **Set Horspool**, can also be considered as a simplification of **Commentz-Walter**.

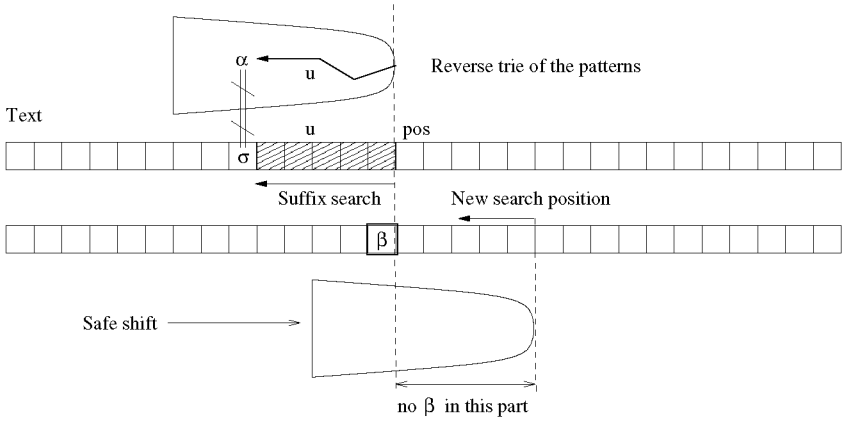

Fig. 3.13. **Horspool** algorithm for a set of patterns. The set is shifted according to the last character of the search window.

The general scheme is shown in Figure 3.13. We start reading the text backwards from a position pos initialized to ℓmin to avoid skipping any occurrence. We read these characters in the trie built on the reverse patterns. If we reach a terminal state, we mark an occurrence. When we fail reading the text, we shift the position pos using the first character read (β in the figure). We shift until β is aligned with another β in the trie. If such a β does not exist, we simply shift by ℓmin characters.

The **Set Horspool** algorithm is $O(n \times \ell max)$ time in the worst case. In

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

Set Horspool ($P = \{p^1, p^2, \ldots, p^r\},\ T = t_1 t_2 ... t_n$)

```
1.    Preprocessing
2.        HO ← Trie(P^rv = {(p^1)^rv, ..., (p^r)^rv})
              δ_HO is its transition function
3.        For c ∈ Σ Do d[c] ← ℓmin
4.        For j ∈ 1 ... r Do
5.            For k ∈ 1 ... m_j − 1 Do d[p_k^j] ← min(d[p_k^j], m_j − k)
6.        End of for
7.    Searching
8.        pos ← ℓmin
9.        While pos ≤ n Do
10.           j ← 0, Current ← initial state of HO
11.           While pos − j > 0 AND δ_HO(t_{pos−j}, Current) ≠ θ Do
12.               If Current is terminal Then
13.                   Mark all the occurrences (F(Current), pos)
14.               End of if
15.               Current ← δ_HO(t_{pos−j}, Current)
16.               j ← j + 1
17.           End of while
18.           pos ← pos + d[t_pos]
19.       End of while
```

Fig. 3.14. **Horspool** algorithm for a set of patterns. The shift is obtained with the first character t_{pos} read.

general, it is only efficient for a very small number of patterns on a relatively large alphabet.

Example using English We search for the set $P = \{$`announce, annual, annually`$\}$ in the text "`CPM_annual_conference_announce`". The trie of the reverse patterns is shown in Figure 3.15

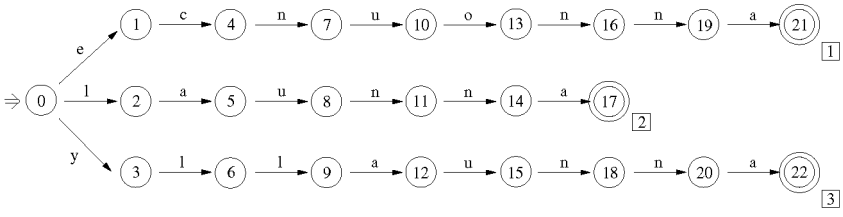

Fig. 3.15. Trie for the reverse set of $P =$ {`announce, annual, annually`}, $P^{rv} =$ {`ecnuonna, launna, yllaunna`}. Double-circled states are terminal.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

$\ell min = 6$,

$d = \left\{ \begin{array}{|c|c|c|c|c|c|c|c|c|} \hline \texttt{a} & \texttt{c} & \texttt{e} & \texttt{l} & \texttt{n} & \texttt{o} & \texttt{u} & \texttt{y} & * \\ \hline 1 & 1 & 6 & 1 & 2 & 4 & 2 & 6 & 6 \\ \hline \end{array} \right.$

1. `CPM_a` [`n`] `nual_conference_announce`
   `n` $\notin \{\texttt{e}, \texttt{l}, \texttt{y}\}$, $d[\texttt{n}] = 2$

2. `CPM_ann` [`u`] `al_conference_announce`
   `u` $\notin \{\texttt{e}, \texttt{l}, \texttt{y}\}$, $d[\texttt{u}] = 2$

3. `CPM_annua` [`l`] `_conference_announce`
   We read in the trie `l`, `a`, `u`, `n`, `n`, `a`. We reach the terminal state 17 and mark an occurrence of $F(17) \rightarrow$ `annual`.
   We re-use the first character read, $d[\texttt{l}] = 1$.

4. `CPM_annual` [`_`] `conference_announce`
   `_` $\notin \{\texttt{e}, \texttt{l}, \texttt{y}\}$, $d[\_] = 6$

5. `CPM_annual_confe` [`r`] `ence_announce`
   `r` $\notin \{\texttt{e}, \texttt{l}, \texttt{y}\}$, $d[\texttt{r}] = 6$

6. `CPM_annual_conference_` [`a`] `nnounce`
   `a` $\notin \{\texttt{e}, \texttt{l}, \texttt{y}\}$, $d[\texttt{a}] = 1$

7. `CPM_annual_conference_a` [`n`] `nounce`
   `n` $\notin \{\texttt{e}, \texttt{l}, \texttt{y}\}$, $d[\texttt{n}] = 2$

8. `CPM_annual_conference_ann` [`o`] `unce`
   `o` $\notin \{\texttt{e}, \texttt{l}, \texttt{y}\}$, $d[\texttt{o}] = 4$

9. `CPM_annual_conference_announc` [`e`]
   We read in the trie `e`, `c`, `n`, `u`, `o`, `n`, `n`, `a`. We reach the terminal state 21 and mark an occurrence of $F(21) \rightarrow$ `announce`.

Example using DNA We search for the set of strings $P =$ {`ATATATA`, `TATAT`, `ACGATAT`} in the text `AGATACGATATATAC`. The trie of the reverse patterns is shown in Figure 3.16

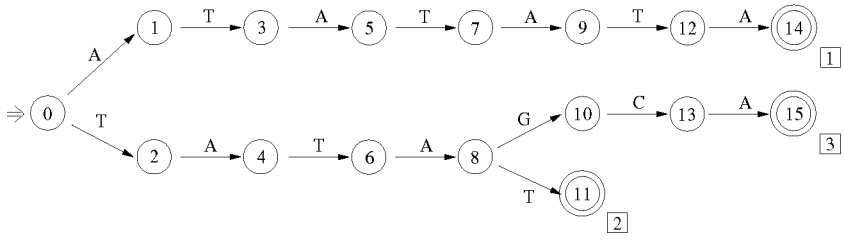

Fig. 3.16. Trie for the reverse set of $P =$ {`ATATATA`, `TATAT`, `ACGATAT`}, $P^{rv} =$ {`ATATATA`, `TATAT`, `TATAGCA`}. Double-circled states are terminal.

$\ell min = 5$,

$\delta = \left\{ \begin{array}{|c|c|c|c|c|} \hline \texttt{A} & \texttt{C} & \texttt{G} & \texttt{T} & * \\ \hline 1 & 5 & 4 & 1 & 5 \\ \hline \end{array} \right.$

1. `AGAT` [`A`] `CGATATATAC`
   We read in the trie `A`, `T`, `A`, and we fail on the next `G`. We re-use the last character of the window, $d[\texttt{A}] = 1$.

2. `AGATA` [`C`] `GATATATAC`
   `C` $\notin \{\texttt{A}, \texttt{T}\}$, $d[\texttt{C}] = 5$.

3. `AGATACGATA` [`T`] `ATAC`
   We read in the trie `T`, `A`, `T`, `A`, `G`, `C`, `A`. We reach the terminal state 15 and mark an occurrence of $F(15) \rightarrow$ `ACGATAT`.
   We re-use the last character of the window, $d[\texttt{T}] = 1$.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

4. AGATACGATAT [A] TAC
   We read in the trie A, T, A, T, A, and we fail on the next G.

   We re-use the first character read, d[A] = 1.

5. AGATACGATATA [T] AC
   We read in the trie T, A, T, A, T. We reach the terminal state 11 and mark an occurrence of $F(11) \rightarrow$ TATAT.

   We re-use the first character read, d[T] = 1.

6. AGATACGATATAT [A] C
   We read in the trie A, T, A, T, A, T, A. We reach the terminal state 14 and mark an occurrence of $F(14) \rightarrow$ ATATATA.

   We re-use the first character read, d[A] = 1.

7. AGATACGATATATA [C]
   C $\notin$ {A, T}, d[C] = 5. Then $pos > n$ and we stop the search.

3.3.3 Wu-Manber algorithm

The poor performance of the extension of **Horspool** to search a set of patterns is a direct consequence of the fact that the lengths of the shifts are usually decreasing, due to the high probability of finding each character of the alphabet in one of the strings.

The algorithm of Wu and Manber [WM94] bypasses this obstacle by reading blocks of characters, which reduces the probability that each block appears in one of the patterns. We consider blocks of length B. The difficulty is that there could be $|\Sigma|^B$ different blocks, requiring too much memory if B becomes large.

Wu and Manber overcome this problem by hashing all the possible blocks using a function h_1 into a limited size table *SHIFT*. Two blocks B_1 and B_2 can be associated with the same position in *SHIFT*. If we consider that for each new position we are reading a block Bl instead of the last character of the **Horspool** algorithm, then the shift given by Bl, $SHIFT(h_1(Bl))$, must be safe. To guarantee this, we save in $SHIFT(j)$ the minimum of the shifts of the blocks Bl such that $j = h_1(Bl)$. More precisely, the table *SHIFT* is built in the following way:

- If a block Bl does not appear in any string of P, we can safely shift $\ell min - B + 1$ characters to the right. Hence we initialize the table by placing $\ell min - B + 1$ everywhere.
- If Bl appears in one of the strings of P, we find its rightmost occurrence ending in j in a string p^i, and set the value of $SHIFT(h_1(Bl))$ to $m_i - j$. To compute all the values of the table *SHIFT*, we consider separately each $p^i = p^i_1 \dots p^i_{m_i}$. For each block $B = p^i_{j-B+1} \dots p^i_j$, we find its corresponding cell $h_1(B)$ in *SHIFT*, and we place in $SHIFT(h_1(B))$ the minimum between the previous value and $m_i - j$.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

```
Wu-Manber(P = {p^1, p^2, ..., p^r},  T = t_1 t_2 ... t_n)
1.    Preprocessing
2.        Computation of B
3.        Construction of the hash tables SHIFT and HASH
4.    Searching
5.        pos ← ℓmin
6.        While pos ≤ n Do
7.            i ← h_1(t_{pos-B+1} ... t_{pos})
8.            If SHIFT[i] = 0 Then
9.                list ← HASH[ h_2(t_{pos-B+1} ... t_{pos}) ]
10.               Verify all the patterns in list one by one against the text
11.               pos ← pos+1
12.           Else pos ← pos + SHIFT[i]
13.           End of if
14.       End of while
```

Fig. 3.17. **Wu-Manber** algorithm for searching a set of strings.

The size B varies with ℓmin, with the number of patterns, and with the size of the alphabet. Wu and Manber show that the value $B = \log_{|\Sigma|}(2 \times \ell min \times r)$ yields the best experimental results. The size of the table *SHIFT* can also vary with the memory space available.

We can shift the search position along the text as long as the value of the shift is strictly positive. When the shift is zero, the text to the left of the search position may be one of the pattern strings. In this case Wu and Manber use a new hash table *HASH*. Each position $HASH(j)$ contains a list of all the strings whose last block is hashed to j by a second hash function h_2. This table permits us to select from P a subset of strings whose last block maps the block Bl read in the text.

For the search, similarly to the **Set Horspool** algorithm, we slide a position *pos* along the text, reading backwards a block Bl of B characters. The position *pos* is initialized to ℓmin. If $j = SHIFT(h_1(Bl)) > 0$, then we shift the window to $pos + j$ and continue the search. Otherwise, $SHIFT(h_1(Bl)) = 0$ and we select a set of strings using *HASH* that we compare to the text. Pseudo-code is given in Figure 3.17.

The original description of the algorithm [WM94] is quite fuzzy. Nothing is given in the article that permits you to calculate the best size of the tables *SHIFT* and *HASH*. Likewise, the hash functions are not specified. All these parameters affect the complexity. In practice, well parametrized, this algorithm is very fast. It is implemented in *Agrep* (Section 7.1.2).

We now present our two running examples. The **Wu-Manber** algorithm uses many hash functions and tables that are difficult to represent. We have chosen some that do not correspond to a real example, but permit us to show the interesting cases. We let $B = 2$ in the following tables.

Example using English We search for the set $P = \{$`announce, annual, annually`$\}$ in the text "`CPM_annual_conference_announce`".

$SHIFT[Bl] =$

string	ll	no ou	an	un nc	ua al	ly	nn nu	ce	*
shift	1	3	4	1	0	0	2	0	5

$HASH[Bl] =$

string	ce ly	ua al	*
string number in P	3, 1	2	$\emptyset$

1. `CPM_` `an` `nual_conference_announce`
   $SHIFT$[`an`] $= 4$.

2. `CPM_annu` `al` `_conference_announce`
   $SHIFT$[`al`] $= 0$. $L = HASH$[`al`] $= \{2\}$.

   We compare p^2 against the text and mark its occurrence. We then shift the search position by 1.

3. `CPM_annua` `l_` `conference_announce`
   $SHIFT$[`l_`] $= 5$.

4. `CPM_annual_con` `fe` `rence_announce`
   $SHIFT$[`fe`] $= 5$.

5. `CPM_annual_conferen` `ce` `_announce`
   $SHIFT$[`ce`] $= 0$. $L = HASH$[`ce`] $= \{3, 1\}$.

   We compare p^1 and p^3 against the text. No string matches. We shift the search position by 1.

6. `CPM_annual_conferenc` `e_` `announce`
   $SHIFT$[`e_`] $= 5$.

7. `CPM_annual_conference_ann` `ou` `nce`
   $SHIFT$[`ou`] $= 3$.

8. `CPM_annual_conference_announ` `ce`
   $SHIFT$[`ce`] $= 0$. $L = HASH$[`ce`] $= \{3, 1\}$.

   We compare p^1 and p^3 against the text. The test succeeds for the string p^1. Hence, we mark its occurrence.

Example using DNA We search the set of strings $P = \{$`ATATATA, TATAT, ACGATAT`$\}$ in the text `AGATACGATATATAC`.

$SHIFT[Bl] =$

string	GA TA	AT	CG GA	AC	*
shift	0	0	3	4	4

$HASH[Bl] =$

string	TA	AT	*
string number in P	1	2, 3	$\emptyset$

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

1. AGA [TA] CGATATATAC
   $SHIFT$[TA] $= 0$. $L = HASH$[TA] $= \{1\}$.

   We compare p^1 against the text. The test fails. We shift the search position by 1.

2. AGAT [AC] GATATATAC
   $SHIFT$[AC] $= 4$.

3. AGATACGA [TA] TATAC
   $SHIFT$[TA] $= 0$. $L = HASH$[TA] $= \{1\}$.

   We compare p^1 against the text. The test fails. We shift the search position by 1.

4. AGATACGAT [AT] ATAC
   $SHIFT$[AT] $= 0$. $L = HASH$[AT] $= \{2, 3\}$.

   We compare p^2 and p^3 against the text. The string p^3 matches. We mark its occurrence. We shift the search position by 1.

5. AGATACGATA [TA] TAC
   $SHIFT$[TA] $= 0$. $L = HASH$[TA] $= \{1\}$.

   We compare p^1 against the text. The test fails. We shift the search position by 1.

6. AGATACGATAT [AT] AC
   $SHIFT$[AT] $= 0$. $L = HASH$[AT] $= \{2, 3\}$.

   We compare p^2 and p^3 against the text. The string p^2 matches. We mark its occurrence. We shift the search position by 1.

7. AGATACGATATA [TA] C
   $SHIFT$[TA] $= 0$. $L = HASH$[TA] $= \{1\}$.

   We compare p^1 against the text. The string p^1 matches. We mark its occurrence. We shift the search position by 1.

8. AGATACGATATAT [AC]
   $SHIFT$[AC] $= 4$.

3.4 Factor based approach

The general factor based approach can be extended directly to a set of strings. We search backwards for the longest suffix u of the text that is also a factor of one of the strings in P. If we fail on a letter σ, then σu is not a factor in any of the strings; thus no string of P can overlap σu.

There are, however, two technical difficulties to overcome. The first problem is to shift the set of patterns safely to avoid skipping an occurrence; the second difficulty is to recognize the factors of a set of strings.

The first two factor based algorithms were the **Dawg-Match** [CCG+93, CCG+99] and the **MultiBDM** [CR94, Raf97]. They were developed with the aim of obtaining fast algorithms on average with good worst-case complexity. Indeed, they are all worst-case linear in the size of the text. But they are inherently complicated, and in practice their performance is poor.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

As we aim to present the simplest and most efficient algorithms, we will not describe these two.

The two algorithms left that use the factor based approach are **Set Backward Dawg Matching** (**SBDM**) and **Set Backward Oracle Matching** (**SBOM**) [AR99]. They extend **BDM** and **BOM**, respectively (Chapter 2). We present the **SBDM** idea, on which **SBOM** is based.

Bit-parallelism is only valuable for a small set of strings. However, similarly to the **Multiple Shift-And** (Section 3.2.1), it permits efficient extended string matching (Chapter 4) and also approximate matching (Chapter 6). We present it first.

3.4.1 Multiple BNDM algorithm

The use of bit-parallelism to search a set of strings $P = \{p^1, \ldots, p^r\}$ is efficient for sets such that $r \times \ell min$ fits in a few computer words [NR00]. For simplicity we assume below that $r \times \ell min \leq w$.

To perform longer shifts, we keep only the prefixes of size ℓmin of the patterns. If we match a prefix, we directly verify the entire string against the text.

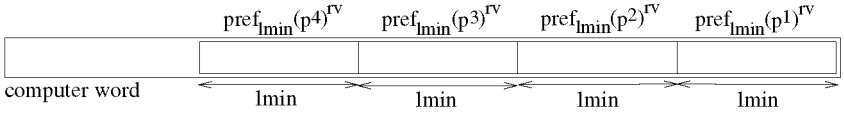

Fig. 3.18. **Multiple BNDM** algorithm. The total $r \times \ell min$ has to fit in w. The notation $pref_{\ell min}(p^i)$ denotes the prefix of size ℓmin of p^i.

The prefixes are packed together as in Figure 3.18 and the search is similar to **BNDM** (Section 2.4.2), with the search performed for all the prefixes at the same time. The only difference is that we need to clear some bits after a shift. The mask CL in Figure 3.19 does that. It prevents the bits used to search for p^i from interfering with those used for p^{i+1}. The variable *last* is still used, but in this case it represents the position where a prefix of one of the strings begins. Pseudo-code of the whole algorithm is shown in Figure 3.19.

Example using English We search the text "`CPM_annual_conference_announce`" for the set of strings $P = \{$`announce`, `annual`, `annually`$\}$.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

Multiple BNDM ($p = p_1 p_2 \dots p_m,\ T = t_1 t_2 \dots t_n$)

```
1.  Preprocessing
2.      For c ∈ Σ Do B[c] ← 0^|P|
3.      ℓ ← 0
4.      For k ∈ 1...r Do
5.          ℓ ← ℓ + ℓmin
6.          For j ∈ 1...ℓmin Do B[p_j^k] ← B[p_j^k] | 0^(|P|−ℓ+j−1) 10^(ℓ−j)
7.      End of for
8.      CL ← (1^(ℓmin−1) 0)^r
9.      DF ← (10^(ℓmin−1))^r
10. Searching
11.     pos ← 0
12.     While pos ≤ n − m Do
13.         j ← ℓmin, last ← ℓmin
14.         D = 1^|P|
15.         While D ≠ 0^|P| Do
16.             D ← D & B[t_(pos+j)]
17.             j ← j − 1
18.             If D & DF ≠ 0^|P| Then
19.                 If j > 0 Then last ← j
20.                 Else        /* at least one prefix matches */
21.                     Check which prefixes of length ℓmin match
                            p^i needs to be checked if
                                D & 0^(|P|−ℓmin×i) 10^(ℓmin−1) 0^(ℓmin×(i−1)) ≠ 0^|P|
22.                     Verify the corresponding string(s) against the text
23.                     Report the occurrence(s) at pos + 1
24.                 End of if
25.             End of if
26.             D ← (D << 1) & CL      /* Shifting and cleaning */
27.         End of while
28.         pos ← pos + last
29.     End of while
```

Fig. 3.19. Bit-parallel code for the **Multiple BNDM** algorithm.

$$B = \left\{ \begin{array}{|c|c|} \hline \texttt{a} & 1\,0\,0\,0\,1\,0\,1\,0\,0\,0\,1\,0\,1\,0\,0\,0\,0\,0 \\ \hline \texttt{l} & 0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0 \\ \hline \texttt{n} & 0\,1\,1\,0\,0\,0\,0\,1\,1\,0\,0\,0\,0\,1\,1\,0\,0\,1 \\ \hline \texttt{o} & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0 \\ \hline \texttt{u} & 0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,1\,0 \\ \hline * & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ \hline \end{array} \right.$$

$DF = 1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0$
$CL = 1\,1\,1\,1\,1\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,1\,1\,0$

1. `CPM_an` `nual_conference_announce`
   $last \leftarrow 6$.
   $D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[n]	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
D =	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
	1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 0
B[a]	1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0
D =	1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0

$D\ \&\ DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 4$

	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
B[_]	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
D =	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

2. CPM_ annual _conference_announce
   last ← 6.
   $D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[1]	0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0
$D =$	0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0
	0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
B[a]	1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0
$D =$	0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
	0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
B[u]	0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0
$D =$	0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
	0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
B[n]	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
$D =$	0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
	0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
B[n]	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
$D =$	0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
	1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
B[a]	1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0
$D =$	1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0

   D & $DF \neq 0^{|P|}$ and $j = 0$, so we check the patterns "annual" and "annually" against the text and mark the occurrence of "annual".

3. CPM_annual _confe rence_announce
   last ← 6.
   $D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[e]	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

4. CPM_annual_confe rence_ announce
   last ← 6.
   $D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[_]	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

5. CPM_annual_conference_ announ ce
   last ← 6.
   $D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[n]	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
$D =$	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
	1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1 0
B[u]	0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
B[o]	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
B[n]	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
B[n]	0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 1
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
B[a]	1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

   D & $DF \neq 0^{|P|}$ and $j = 0$, so we check the string "announce" against the text and mark its occurrence.

   The next shift of the search window gives $pos > n - \ell min$ and the search stops.

Example using DNA We search for the set of strings $P = \{$ATATATA, TATAT, ACGATAT$\}$ in the text AGATACGATATATAC.

$$B = \left\{ \begin{array}{|c|c|} \hline \texttt{A} & 1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,0\,1\,0 \\ \hline \texttt{C} & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0 \\ \hline \texttt{G} & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0 \\ \hline \texttt{T} & 0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,0\,0\,0\,1 \\ \hline \texttt{*} & 0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0 \\ \hline \end{array} \right.$$

$$\begin{array}{lcl} DF & = & 1\,0\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\,0 \\ CL & = & 1\,1\,1\,1\,0\,1\,1\,1\,1\,0\,1\,1\,1\,1\,0 \end{array}$$

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

1. AGATA CGATATATAC

$last \leftarrow 5$.
$D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[A]	1 0 1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 1 0 1 0 1 0 1 0 0 1 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 4$

	0 1 0 1 0 1 0 1 0 0 0 0 1 0 0
B[T]	0 1 0 1 0 1 0 1 0 1 0 0 0 0 1
$D =$	0 1 0 1 0 1 0 1 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 3$

	1 0 1 0 0 0 1 0 0 0 0 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 0 0 1 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 2$

	0 1 0 0 0 1 0 0 0 0 0 0 0 0 0
B[G]	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

2. AG ATACG ATATATAC

$last \leftarrow 5$.
$D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[G]	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
B[C]	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 2$

	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
B[T]	0 1 0 1 0 1 0 1 0 0 0 0 1
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

3. AGAT ACGAT ATATAC

$last \leftarrow 5$.
$D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[T]	0 1 0 1 0 1 0 1 0 1 0 0 0 0 1
$D =$	0 1 0 1 0 1 0 1 0 1 0 0 0 0 1

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 4$

	1 0 1 0 0 0 1 0 1 0 0 0 0 1 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 0 0 1 0 1 0 0 0 0 1 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 3$

	0 1 0 0 0 1 0 1 0 0 0 0 1 0 0
B[G]	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
B[C]	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j = 0$, so we check the pattern ACGATAT against the text and mark its occurrence.

4. AGATACG ATATA TAC

$last \leftarrow 5$.
$D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 1 0 1 0 1 0 0 1 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 4$

	0 1 0 1 0 1 0 1 0 0 0 0 1 0 0
B[T]	0 1 0 1 0 1 0 1 0 1 0 0 0 0 1
$D =$	0 1 0 1 0 1 0 1 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 3$

	1 0 1 0 0 0 1 0 1 0 0 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 0 0 1 0 1 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

$last \leftarrow 2$

	0 1 0 0 0 1 0 0 0 0 0 0 0 0
B[T]	0 1 0 1 0 1 0 1 0 1 0 0 0 1
$D =$	0 1 0 0 0 1 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 1$

	1 0 0 0 0 0 0 0 0 0 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 0 0 0 0 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j = 0$, so we check the string **ATATATA** against the text and mark its occurrence.

5. **AGATACGA** TATAT **AC**

$last \leftarrow 5$.
$D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[T]	0 1 0 1 0 1 0 1 0 0 0 0 0 1
$D =$	0 1 0 1 0 1 0 1 0 0 0 0 0 1

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 4$

	1 0 1 0 0 0 1 0 1 0 0 0 0 1 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 0 0 1 0 1 0 0 0 0 1 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 3$

	0 1 0 0 0 1 0 1 0 0 0 0 1 0 0
B[T]	0 1 0 1 0 1 0 1 0 1 0 0 0 1
$D =$	0 1 0 0 0 1 0 1 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 2$

	1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 0 0 0 0 1 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 1$

	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
B[T]	0 1 0 1 0 1 0 1 0 0 0 0 0 1
$D =$	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j = 0$, so we check the string **TATAT** against the text and mark its occurrence.

6. **AGATACGAT** ATATA **C**

$last \leftarrow 5$.
$D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[A]	1 0 1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 1 0 1 0 1 0 1 0 0 1 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 4$

	0 1 0 1 0 1 0 1 0 0 0 0 1 0 0
B[T]	0 1 0 1 0 1 0 1 0 1 0 0 0 0 1
$D =$	0 1 0 1 0 1 0 1 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 3$

	1 0 1 0 0 0 1 0 0 0 0 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 1 0 0 0 1 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 2$

	0 1 0 0 0 1 0 0 0 0 0 0 0 0 0
B[T]	0 1 0 1 0 1 0 1 0 0 0 0 0 1
$D =$	0 1 0 0 0 1 0 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 1$

	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j = 0$, so we check the string **ATATATA** against the text, but we fail to recognize an occurrence.

7. **AGATACGATA** TATAC

$last \leftarrow 5$.
$D \leftarrow 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1$

	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
B[C]	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
B[A]	1 0 1 0 1 0 1 0 1 0 0 1 0
$D =$	0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0

D & $DF \neq 0^{|P|}$ and $j > 0$, then $last \leftarrow 3$

	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
B[T]	0 1 0 1 0 1 0 1 0 0 0 0 0 1
$D =$	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

3.4.2 Set Backward Dawg Matching idea

The **SBDM** algorithm uses a suffix automaton to recognize backwards the factors in a window of size ℓmin that is shifted along the text.

3.4.2.1 Suffix automaton for a set of strings

The suffix automaton for a set of strings [BBE+87] is an automaton that recognizes the suffixes of the set of strings P it is built on. Let γ be the number of states of the trie built on P ($\gamma \leq |P| + 1$). Then the number of states of the suffix automaton is at least γ and at most 2γ. It is also $O(\gamma)$ in its number of transitions.

The construction algorithm is an extension of the construction for a single string (Section 2.4.1), but this time the resulting automaton is not necessarily minimal. The construction is linear in the size of P, but it is complex and slow.

3.4.2.2 Search algorithm

The suffix automaton is built in $O(r \times \ell min)$ time on $P^{rv}_{\ell min}$, the set of reverse prefixes of length ℓmin of the strings in P [BBE+87]. The search is done through a window of size ℓmin, which we slide along the text. In this window, we read backwards the longest suffix that is also a factor of one of the prefixes of length ℓmin of the strings in P. Two cases may occur.

(i) We fail to recognize a factor, that is, we reach a letter σ that does not correspond to a transition in the suffix automaton of $P^{rv}_{\ell min}$. No other prefix of a string can overlap the part of the window read. We therefore shift the window so that its new starting position corresponds to the position next to σ.

(ii) We reach the beginning of the window in a state q of the suffix automaton. This means that we recognized a prefix $L(q)$ of a string in $F(q)$ (Section 3.1). We then verify a possible occurrence by comparing each string in $F(q)$ against the text. We finally move the search window by 1 and start the search again.

The worst-case complexity of **SBDM** is $O(n \times |P|)$, which is very high. However, for reasonable numbers of strings on a not too small alphabet, this algorithm is sublinear on average. The practical limit of this algorithm is the construction of the suffix automaton. For large sets of strings, it is too slow to be amortized by the time saved on the search phase. Moreover, the memory the suffix automaton requires quickly becomes too large as the set increases. We do not describe this algorithm in depth, nor give a pseudocode, because **SBOM** uses the same approach but overcomes the bottleneck

of the suffix automaton with a lighter and simpler data structure. **SBOM** is faster than **SBDM** in all cases.

Note that the algorithm can be improved using the variable *last*, as was done for **BDM** (Section 2.4.1).

3.4.3 Set Backward Oracle Matching algorithm

The **Set Backward Oracle Matching algorithm** (**SBOM**) [AR99] uses a factor oracle of the set of strings. The factor oracle of P recognizes at least all the factors of the strings in P. The search algorithm is similar to **SDBM**. We slide a window of size ℓmin along the text, reading backward a suffix of the window in the factor oracle. If we fail on a letter σ, we can safely shift the window past σ. If not, we reach the beginning of the window and verify a subset of P against the text.

3.4.3.1 Factor oracle of a set of strings

The factor oracle construction on a set of strings resembles the Aho-Corasick automaton construction. The only difference appears when going down the supply path looking for an outgoing transition labeled by σ. In the Aho-Corasick automaton construction, if this transition does not exist, we just jump to the next state on the supply path (Section 3.2.2). In the factor oracle construction, we create in addition a transition labeled by σ from each state on the supply path to the state where the original transition leads.

More precisely, we begin by building the trie of the set of strings P with the algorithm given in Figure 3.6. The states of the factor oracle are those of the trie as well as the initial state I and the terminal states. Hence, the factor oracle has at most $|P| + 1$ states, including the initial one.

To build the "external transitions," which are at most $|P|$, we associate to each state q a "supply state," computed simultaneously with the new transitions in transversal order. The supply state of the initial state is set to θ.

To explain the construction, we assume that we have already computed the supply function of all the states before state *Current* in transversal order. We consider the parent *Parent* of *Current* in the trie, leading to *Current* by σ, that is, $Current = \delta_{OR}(Parent, \sigma)$. The supply state $S_{OR}(Parent)$ has already been computed, and we go down the supply function from state $S_{OR}(Parent)$. We use a variable *Down* initialized to $S_{OR}(Parent)$ and we repeat the following steps.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

ST_1 If $Down = \theta$, then $S_{OR}(Current) \leftarrow I$.

ST_2 If $Down \neq \theta$ and there does not exist a transition from $Down$ labeled by σ, then build a transition from state $Down$ to state $Current$ by σ and return to step ST_1 with $Down \leftarrow S_{OR}(Down)$.

ST_3 If $Down \neq \theta$ and there exists a transition from $Down$ labeled by σ leading to a state Im, then set $S_{OR}(Current) \leftarrow Im$ and stop processing state $Current$.

The resulting factor automaton recognizes at least all factors of P [AR99]. The construction algorithm is worst-case time $O(|P|)$. Its pseudo-code is given in Figure 3.20.

Build_Oracle_Multiple($P = \{p^1, p^2, \ldots, p^r\}$)

1. $OR\_trie \leftarrow$ **Trie**(P)
   δ_{OR} is its transition function
2. Mark the states that correspond to an entire string p^i as terminal
3. $I \leftarrow$ root of OR_trie
4. $S_{OR}(I) \leftarrow \theta$
5. **For** $Current$ in transversal order **Do**
6. $Parent \leftarrow$ parent in OR_trie of $Current$
7. $\sigma \leftarrow$ label of the transition from $Parent$ to $Current$
8. $Down \leftarrow S_{OR}(Parent)$
9. **While** $Down \neq \theta$ AND $\delta_{OR}(Down, \sigma) = \theta$ **Do**
10. $\delta_{OR}(Down, \sigma) \leftarrow Current$
11. $Down \leftarrow S_{OR}(Down)$
12. **End of while**
13. **If** $Down \neq \theta$ **Then**
14. $S_{OR}(Current) \leftarrow \delta_{OR}(Down, \sigma)$
15. **Else** $S_{OR}(Current) \leftarrow I$
16. **End of if**
17. **End for**

Fig. 3.20. Construction of the factor oracle for a set $P = \{p^1, p^2, \ldots, p^r\}$. The state $Current$ goes through the trie OR_trie built on P in transversal order. The state $Down$ goes down the supply links from the parent of $Current$ looking for an outgoing transition labeled with the same character as between $Current$ and its parent, creating it if it does not exist.

3.4.3.2 Search with the factor oracle

The factor oracle is built in $O(r \times \ell min)$ time on the reverse prefixes of length ℓmin of the strings in P. The search is done through a window of size ℓmin, which we slide along the text. In this window, we read backwards the longest suffix that labels a path from the initial state. Two cases may occur.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

(i) We fail to recognize a factor, that is, we reach a letter σ that does not correspond to a transition in the factor oracle of $P^{rv}_{\ell min}$. No other prefix of a string can overlap the part of the window read. We therefore shift the window so that its new starting position corresponds to the position next to σ.

(ii) We reach the beginning of the window in a state q of the factor oracle. When using a suffix automaton in **SBDM**, we can be sure at this step that we recognized a prefix of one of the strings. However, the factor oracle accepts paths of size ℓmin ending in terminal states that do not correspond to any prefix. Hence, we have to verify first that we read the prefix $L(q)^{rv}$ and only if this is the case we verify a possible occurrence by comparing each string in $F(q)$ against the text. We finally move the search window by 1 and start the search again.

SBOM($P = \{p^1, p^2, \ldots, p^r\}$, $T = t_1 t_2 \ldots t_n$)

1. `Preprocessing`
2. $\ell min \leftarrow$ minimal length of strings in $p^i \in P$
3. $Or \leftarrow$ **Build_Oracle_Multiple**($\{pref_{\ell min}(p^1)^{rv}, pref_{\ell min}(p^2)^{rv}$ $\ldots, pref_{\ell min}(p^r)^{rv}\}$)
   δ_{Or} is its transition function
4. **For** q state of Or **Do** $F(q) \leftarrow \emptyset$
5. **For** $i \in 1 \ldots r$ **Do**
6. $F(q) \leftarrow F(q) \cup \{i\}$, where q is the state reached by $pref_{\ell min}(p^i)^{rv}$
7. **End of for**
8. `Searching`
9. $pos \leftarrow 0$
10. **While** $pos \leq n - \ell min$ **Do**
11. $Current \leftarrow$ initial state of Or
12. $j \leftarrow \ell min$
13. **While** $j \geq 1$ AND $Current \neq \theta$ **Do**
14. $Current \leftarrow \delta_{Or}(Current, t_{pos+j})$
15. $j \leftarrow j - 1$
16. **End of while**
17. **If** $Current \neq \theta$ AND $j = 0$ AND $T_{pos+1 \ldots pos+\ell min} = L(Current)^{rv}$ **Then**
18. Verify all the patterns in $F(Current)$ one by one against the text
19. $j \leftarrow 1$
20. **End of if**
21. $pos \leftarrow pos + j$
22. **End of while**

Fig. 3.21. Pseudo-code for the **SBOM** algorithm. The notation $pref_{\ell min}(p^i)$ denotes the prefix of size ℓmin of the string p^i.

Pseudo-code for **SBOM** is given Figure 3.21. Its worst-case complexity is $O(n \times |P|)$, the same as **SBDM**. However, this algorithm is sublinear

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

on average. The construction of the factor oracle is fast and requires little memory, which permits using this algorithm to search large sets of strings on relatively small texts.

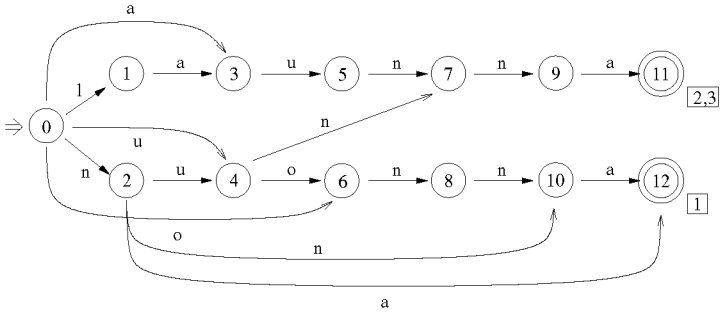

Fig. 3.22. Factor oracle for the reverse set of $P_{\ell min} = \{\texttt{announ}, \texttt{annual}\}$. Double-circled states are terminal.

Example using English We search the text "`CPM_annual_conference_announce`" for the set of strings $P = \{\texttt{announce}, \texttt{annual}, \texttt{annually}\}$. The factor oracle of the reverse set of $P_{\ell min} = \{\texttt{announ}, \texttt{annual}\}$ is shown in Figure 3.22.

1. `CPM_an` `nual_conference_announce`

   We read `n`, `a` in the factor oracle. We fail on the next _. We then shift the window after "_".

2. `CPM_` `annual` `_conference_announce`

   We read `l`, `a`, `u`, `n`, `n`, `a` in the factor oracle. We reach the beginning of the window in state 11. We compare the strings $F(11) \rightarrow$ `annual`, `annually`. We mark an occurrence of "`annual`". We then shift the window by 1.

3. `CPM_a` `nnual_` `conference_announce`

   We fail reading "_" in the oracle. We shift the window after "_".

4. `CPM_annual_` `confer` `ence_announce`

   We fail reading `r` in the oracle. We shift the window after `r`.

5. `CPM_annual_confer` `ence_a` `nnounce`

   We read `a` in the oracle, but we fail on the next letter "_". We shift the window after "_".

6. `CPM_annual_conference_` `announ` `ce`

   We read `n`, `u`, `o`, `n`, `n`, `a` in the factor oracle. We reach the beginning of the window in state 12. We compare the strings $F(12) \rightarrow$ `announce`. We mark an occurrence of "`announce`". We then shift the window by 1.

7. `CPM_annual_conferencea_` `nnounc` `e`

   We fail reading `c` in the oracle. We shift the window after `c`. Then $pos > n - \ell min$ and the search stops.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

Example using DNA We search for the set of strings $P = \{\texttt{ATATATA}, \texttt{TATAT}, \texttt{ACGATAT}\}$ in the text `AGATACGATATATAC`. The factor oracle for the reverse set of $P^{\ell min} = \{\texttt{ATATA}, \texttt{TATAT}, \texttt{ACGAT}\}$ is shown in Figure 3.23.

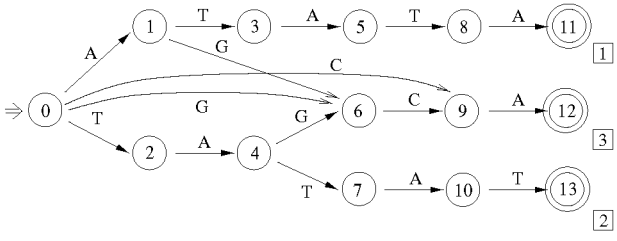

Fig. 3.23. Factor oracle for the reverse set of $P_{\ell min} = \{\texttt{ATATA}, \texttt{TATAT}, \texttt{ACGAT}\}$. Double-circled states are terminal.

1. `AGATA` | `CGATATATAC`

   We read `A`, `T`, `A` in the factor oracle, and we fail on the next `G`. We then shift the window after `G`.

2. `AG` | `ATACG` | `ATATATAC`

   We read `G`, `C`, `A` in the factor oracle, and we fail on the next `T`. We then shift the window after `T`.

3. `AGAT` | `ACGAT` | `ATATAC`

   We read `T`, `A`, `G`, `C`, `A` in the factor oracle. We reach the beginning of the window in state 12. We compare the strings $F(12) \rightarrow$ `ACGATAT`. We mark an occurrence and shift the window by 1.

4. `AGATA` | `CGATA` | `TATAC`

   We read `A`, `T`, `A` in the factor oracle, and we fail on the next `G`. We then shift the window after `G`.

5. `AGATACG` | `ATATA` | `TAC`

   We read `A`, `T`, `A`, `T`, `A` in the factor oracle. We reach the beginning of the window in state 11. We compare the string $F(11) \rightarrow$ `ATATATA`. We mark an occurrence and shift the window by 1.

6. `AGATACGA` | `TATAT` | `AC`

   We read `T`, `A`, `T`, `A`, `T` in the factor oracle. We reach the beginning of the window in state 13. We compare the string $F(13) \rightarrow$ `TATAT`. We mark an occurrence and shift the window by 1.

7. `AGATACGAT` | `ATATA` | `C`

   We read `A`, `T`, `A`, `T`, `A` in the factor oracle. We reach the beginning of the window in state 11. We compare the string $F(11) \rightarrow$ `ATATATA` and fail. We shift the window by 1.

8. `AGATACGATA` | `TATAC`

   We read `C`, `A` in the factor oracle and fail on the next `T`. We shift the window and the search stops since $pos > n - \ell min$.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

3.5 Experimental maps

We present in this section some maps of efficiency for the different multiple string matching algorithms, showing for all of them the zone in which they are most efficient in practice. The text of 10 megabytes is randomly built, as are the patterns. The experiments were performed on a $w = 32$ bits Ultra Sparc 1 running SunOs 5.6. The sets contain 5, 10, 100, and 1000 strings of the same length, varying from 5 to 100 in steps of 5. We tested all the algorithms presented. The **Wu-Manber** algorithm used in these experiments is the implementation found in *Agrep*. Its performance may vary, depending on the hash functions and the sizes of the tables used.

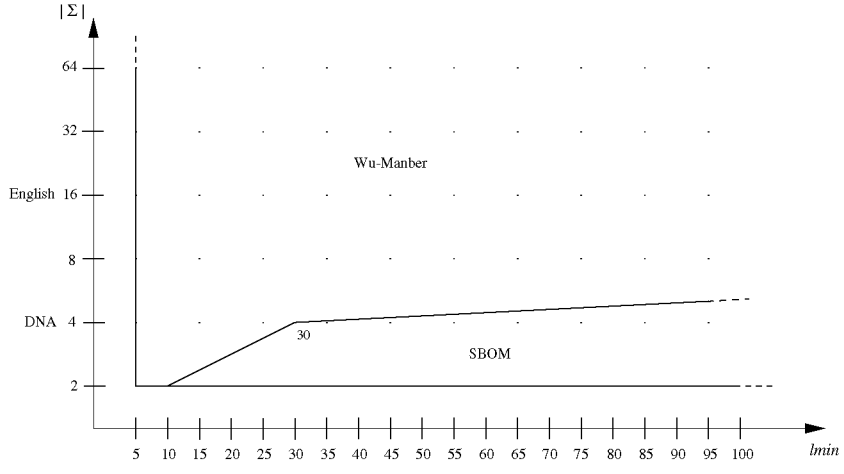

Fig. 3.24. Map of the most efficient algorithms when searching for 5 strings.

The maps are shown in Figures 3.24 to 3.27. The most efficient algorithms are just **Wu-Manber**, the advanced **Aho-Corasick**, and **SBOM**. As the set grows in size, **SBOM** becomes more and more attractive. The advanced **Aho-Corasick** also improves in comparison with the others for short strings, since it reads the text only once.

3.6 Other algorithms and references

Dynamic multiple string matching The algorithms presented in this chapter preprocess a fixed set of strings (a dictionary) in order to perform the search. However, if we need to modify the dictionary by adding or removing a string, we need to preprocess the new dictionary from scratch. The problem of searching for a set of strings in a text and allowing efficient modifications of the set is called *dynamic string matching*. It has been solved

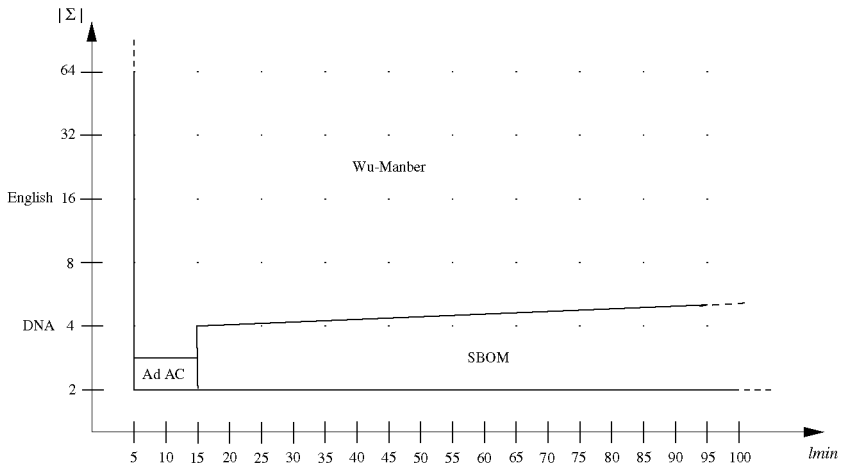

Fig. 3.25. Map of the most efficient algorithms when searching for 10 strings.

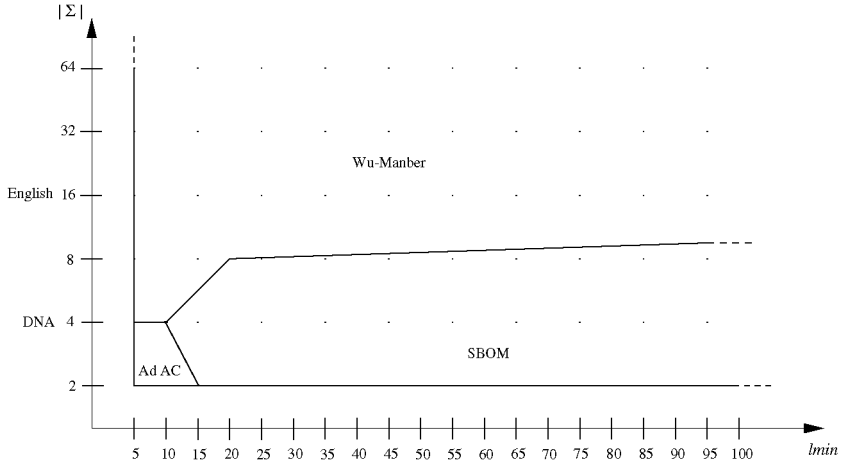

Fig. 3.26. Map of the most efficient algorithms when searching for 100 strings.

recently [SV96] with optimal worst-case complexities: (i) preprocessing of the set of strings in $O(|P|)$ time; (ii) adding or removing a string p in $O(|p|)$ time; and (iii) finding all occurrences of P in the text in $O(n + nocc)$ time, where $nocc$ is the number of occurrences of P in the text.

An application of dynamic string matching is in the matching of a set of strings with variable length "don't cares" [KR95].

On the Commentz-Walter algorithm Several variations of the **Commentz-Walter** algorithm have been designed to limit its worst-case com-

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press

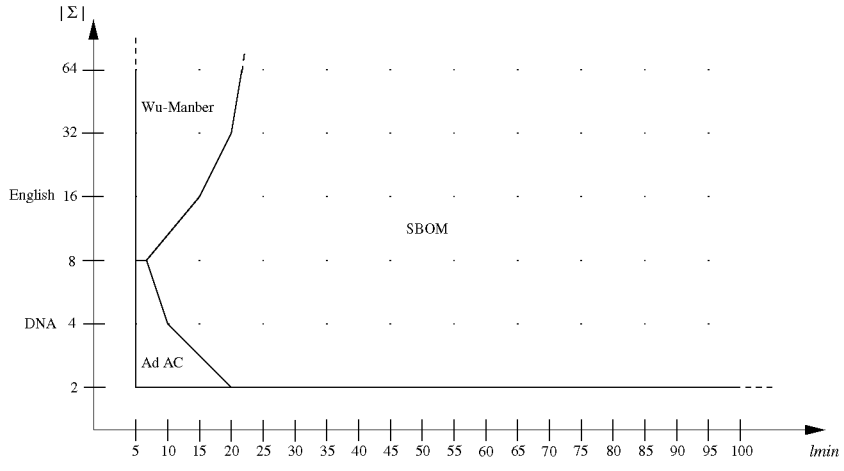

Fig. 3.27. Map of the most efficient algorithms when searching for 1000 strings.

plexity [Sri86] by using additional memory. These algorithms are, however, not efficient in practice.

On matching a set of strings on unbounded alphabets The problem of matching a set of strings of the same length m on an unbounded alphabet has been investigated [Bre95]. The resulting algorithm runs in $O((\log(|P|)/m + 1) \times n)$ comparisons after an $O(|P| \times m \times \log|A|)$ preprocessing time, where A is the alphabet on which the set P is built.

https://doi.org/10.1017/CBO9781316135228.003 Published online by Cambridge University Press