2
String matching

2.1 Basic concepts

The string matching problem is that of finding all the occurrences of a given pattern $p = p_1 p_2 \ldots p_m$ in a large text $T = t_1 t_2 \ldots t_n$, where both T and p are sequences of characters from a finite character set Σ. Given strings x, y, and z, we say that x is a prefix of xy, a suffix of yx, and a factor of yxz.

Many algorithms exist to solve this problem. The oldest and most famous are the **Knuth-Morris-Pratt** and the **Boyer-Moore** algorithms. These algorithms appeared in 1977. The first is worst-case linear in the size of the text. This $O(n)$ complexity is a lower bound for the worst case of any string matching algorithm. The second is $O(mn)$ in the worst case but is sublinear on average, that is, it may avoid reading some characters of the text. An $O(n \log_{|\Sigma|} m/m)$ lower bound on the average complexity has been proved in [Yao79].

Since 1977, many studies have been undertaken to find simpler algorithms, optimal average-case algorithms, algorithms that could also search extended patterns, constant space algorithms, and so on. There exists a large variety of research directions that have been tried, many of which lead to different string matching algorithms.

The aim of this chapter is not to present as many algorithms as possible, nor to give an exhaustive list of them. Instead, we will present the most efficient algorithms, which means the algorithms that for some pattern length and some alphabet size yield the best experimental results. Among those that have more or less the same efficiency, we will present the simplest.

The algorithms we present derive from three general search approaches, according to the way the text is searched. For all of them, a *search window* of the size of the pattern is slid from left to right along the text, and the pattern is searched for inside the window. The algorithms differ in the way in which the window is shifted. The general scheme is shown Figure 2.1, together with our favorite running example on English, which we will show for all our algorithms.

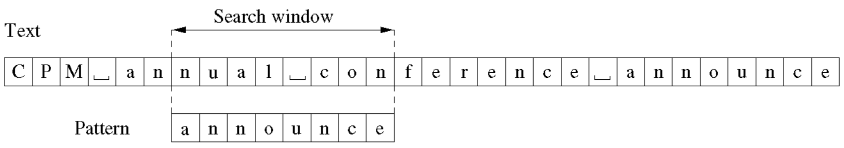

Fig. 2.1. The search is done in a window that slides along the text. The search window has the size of the pattern.

In general, strings that are searched for in natural language texts are simpler than in DNA sequences because the former contain fewer intrinsic repetitions. To show all the tricky cases that could appear, we also show the behavior of all our algorithms when searching for the string `ATATA` in the sequence `AGATACGATATATAC`.

The three search approaches are presented below.

Prefix searching (Figure 2.2) The search is done forward in the search window, reading all the characters of the text one after the other. For each position of the window, we search for the longest prefix of the window that is also a prefix of the pattern. The **Knuth-Morris-Pratt** algorithm uses this approach.

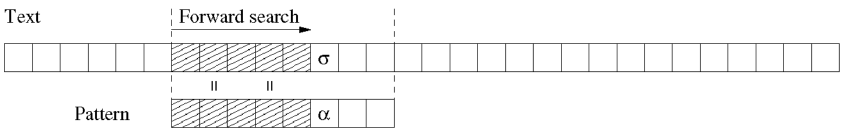

Fig. 2.2. First approach: We search for a prefix of the pattern in the current window.

Suffix searching (Figure 2.3) The search is done backwards along the search window, reading the longest suffix of the window that is also a suffix of the pattern. This approach enables us on average to avoid reading some characters of the text, and therefore leads to sublinear average-case algorithms. The most famous algorithm using this technique is the **Boyer-Moore** algorithm, which has been simplified by Horspool and by Sunday.

Factor searching (Figure 2.4) The search is done backwards in the search window, looking for the longest suffix of the window that is also

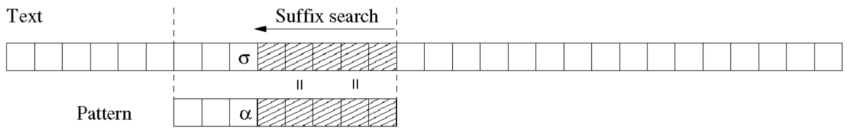

Fig. 2.3. Second approach: We search for a suffix of the pattern in the current window.

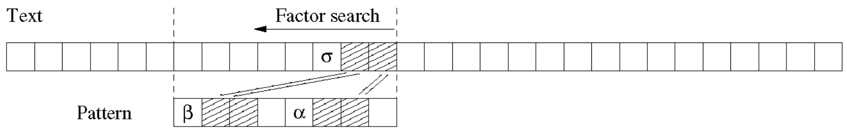

Fig. 2.4. Third approach: We search for a factor of the pattern in the current window.

a factor of the pattern. As with suffix searching, this approach leads to sublinear expected algorithms, and even to optimal algorithms. The main drawback is that it requires a way to recognize the set of factors of the pattern, and this is quite complex.

These three approaches lead to algorithms that are efficient in several cases, depending on the size of the pattern and the size of the alphabet. An experimental map of their relative performances is given in Section 2.5.

2.2 Prefix based approach

Suppose that we have read the text up to position i and that we know the length of the longest suffix of the text read that corresponds to a prefix of the pattern p. When this length is $|p|$ we have an occurrence. The main algorithmic problem is to find an efficient way to compute this length when we read the next character of the text. There exist two classical ways to solve this problem:

- The first is to find a mechanism to effectively compute the longest suffix of the text read that is also a prefix of p, preferably in amortized constant time per character. This is what the algorithm of Knuth, Morris, and Pratt, **KMP**, does [KMP77].
- The second is to maintain a kind of set of all the prefixes of p that are also suffixes of the text read, and update the set at each character read. The bit-parallelism technique enables managing such a set in an efficient way if the pattern is short enough. This leads to the **Shift-And** and **Shift-Or** algorithms [WM92b, BYG89b].

We do not give pseudo-code for the **Knuth-Morris-Pratt** algorithm, nor a deeper study, for this algorithm improves in practice over suffix or factor searching only for strings of less than 8 characters. In this range, the **Shift-And** or **Shift-Or** algorithms can be run on any computer, and are at least twice as fast and much simpler to implement.

2.2.1 Knuth-Morris-Pratt idea

The **Knuth-Morris-Pratt** (**KMP**) algorithm updates for each text character read the length of the longest prefix of the pattern that is also a suffix of the text. The mechanism is based on the following observation. Let us complete Figure 2.2 of the general prefix matching approach with a representation of what what we would like to obtain. This is shown in Figure 2.5. The string $v\beta$ is a new potential prefix of the pattern that could be the new longest prefix of p that is also a suffix of $t_1 \dots t_{i+1}$. We observe that v is a suffix of u, and also a prefix. We call it a *border* of u. Also, the character β has to be equal to t_{i+1} (σ on the figure).

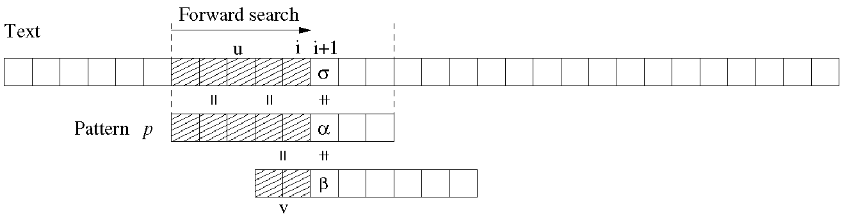

Fig. 2.5. The shift in the **Knuth-Morris-Pratt** algorithm. The string v is a suffix of the prefix u, and also a prefix. The character β differs from α, which differs from the text character σ, on which the prefix search failed.

The original idea, due to Morris and Pratt [MP70], is

- Precompute the longest border $b(u)$ for each prefix u of the pattern.
- Now, in the current position, let u be the longest prefix of p that is a suffix of $t_1 \dots t_i$. We read the character $\sigma = t_{i+1}$ of the text. If $\sigma = p_{|u|+1}$ (denoted α in Figure 2.5), then the new longest prefix is $up_{|u|+1}$. However, if $\sigma \neq p_{|u|+1}$, then we compare σ with $p_{|b(u)|+1}$. If $\sigma = p_{|b(u)|+1}$, then $b(u)p_{|b(u)|+1}$ is the new longest prefix of p that is a suffix of $t_1 \dots t_{i+1}$. If $\sigma \neq p_{|b(u)|+1}$, then we compare it with $\sigma = p_{|b(b(u))|+1}$ and so on, until one border is followed by σ, or until there are no more borders (the empty border ε does not have a border), in which case the new longest prefix is the empty string ε.

Knuth proposed the following improvement. We know that if the comparison of $\sigma = t_{i+1}$ with $p_{|u|+1}$ fails, the letter that follows any border of u must differ from $p_{|u|+1}$ if it is to match σ. So at the precomputing phase, we can precompute for each *proper* prefix u of p ($p = uw$, $w \neq \varepsilon$) the longest border v that satisfies $p_{|u|+1} \neq p_{|v|+1}$.

The **KMP** algorithm is $O(n)$ in the worst and average case for the searching phase. For the preprocessing phase, the goal is to compute two things: first, for each proper prefix u of the pattern, the longest border v such that $p_{|u|+1} \neq p_{|v|+1}$; and second, for the pattern itself, its own longest border. Now, if we read the pattern $p_1 \dots p_m$ character by character, and if we want to compute at each position p_{i+1} the length of the longest border of $p_1 \dots p_{i+1}$, we want, in fact, to compute the longest suffix of $p_1 \dots p_{i+1}$ that is also a prefix of p. It turns out that we are applying the **KMP** algorithm for searching p itself. The preprocessing phase of **KMP** can also be done with **KMP**, and its complexity is $O(m)$.

We do not explain **KMP** further. Details can be found in [KMP77, CR94]. Many studies and variants exist. We give in Section 2.6 the most important bibliographic references.

2.2.2 Shift-And/Shift-Or algorithm

The idea of the **Shift-And** and the **Shift-Or** algorithms is much simpler than that of **KMP**. It consists in keeping a set of all the prefixes of p that match a suffix of the text read. The algorithms use bit-parallelism to update this set for each new text character. This set is represented by a bit mask $D = d_m \dots d_1$.

We first explain the **Shift-And** algorithm, which is easier to explain than **Shift-Or**.

We put a 1 in the j-th position of D (the j-th position of D is said to be *active*) if and only if $p_1 \dots p_j$ is a suffix of $t_1 \dots t_i$. If the size of p is less than w, then this array will fit in a computer register. We report a match whenever d_m is active.

When reading the next text character t_{i+1}, we have to compute the new set D'. A position $j+1$ in this set will be active if and only if the position j was active in D, that is, $p_1 \dots p_j$ was a suffix of $t_1 \dots t_i$ and t_{i+1} matches p_{j+1}. This new set is easy to compute in constant time using bit-parallel operations.

The algorithm first builds a table B, which stores a bit mask $b_m \ldots b_1$ for each character. The mask in $B[c]$ has the j-th bit set if $p_j = c$.

We initially set $D = 0^m$, and for each new text character t_{i+1} we update D using the formula

$$D' \ \leftarrow \ ((D << 1) \mid 0^{m-1}1) \ \ \& \ \ B[t_{i+1}] \qquad (2.1)$$

Intuitively, the "$<<$" shifts the positions to the left to mark at step $i+1$ which positions of p were suffixes at step i. We also mark the empty string ε as a suffix, so we OR the new bit mask with $0^{m-1}1$. Now, we keep from these positions only those such that t_{i+1} matches p_{j+1}, by AND-ing this set of positions with the set $B[t_{i+1}]$ of positions of t_{i+1} in p.

The cost of this algorithm is $O(n)$, assuming that the operations in formula (2.1) can be done in constant time, in practice when the pattern fits in a few computer words.

The **Shift-Or** algorithm is a tricky implementation of **Shift-And**. The idea is to avoid using the "$0^{m-1}1$" mask of formula (2.1) in order to speed up the computation. For this, we complement all the bit masks of B and use a complemented bit mask D. As the shift "$<<$" operation will introduce a 0 to the right of D', the new suffix coming from the empty string is already in D'.

```
Shift-And ($p = p_1p_2 \ldots p_m$, $T = t_1t_2 \ldots t_n$)
1.    Preprocessing
2.        For $c \in \Sigma$ Do $B[c] \leftarrow 0^m$
3.        For $j \in 1 \ldots m$ Do $B[p_j] \leftarrow B[p_j] \mid 0^{m-j}10^{j-1}$
4.    Searching
5.        $D \leftarrow 0^m$
6.        For $pos \in 1 \ldots n$ Do
7.            $D \leftarrow ((D << 1) \mid 0^{m-1}1) \ \& \ B[t_{pos}]$
8.            If $D \ \& \ 10^{m-1} \neq 0^m$ Then report an occurrence at $pos - m + 1$
9.        End of for
```

Fig. 2.6. **Shift-And** algorithm.

The **Shift-And** and the **Shift-Or** algorithms can be seen as the simulation of a nondeterministic automaton that searches for the pattern in the text (Figure 2.7). Formula (2.1) is then related to the moves in the nondeterministic automaton for each new text character: Each state gets the value of the previous state, but only if the text character matches the corresponding arrow.

The "$\mid 0^{m-1}1$" after the shift allows a match to begin at the current text position. This corresponds to the self-loop at the beginning of the automaton.

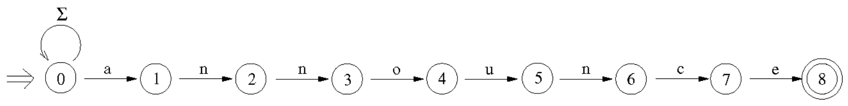

Fig. 2.7. Nondeterministic automaton recognizing all prefixes of the pattern "announce".

The automaton point of view is also valid for **KMP**, which can be seen as an economical method to compute a deterministic automaton that searches for the pattern in the text. The difference between **KMP** and **Shift-Or** is that the former uses a deterministic automaton that the latter simulates with bit-parallelism. However, **Shift-Or** is in practice twice as fast as **KMP**, is simpler to implement, and can handle extended strings (Chapter 4).

Example using English We search for the string "announce" in the text "annual_announce".

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

$D = 0\,0\,0\,0\,0\,0\,0\,0$

1. $\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \text{Reading } \texttt{a} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,1 \end{array}$

2. $\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,1\,1 \\ \text{Reading } \texttt{n} & 0\,0\,1\,0\,0\,1\,1\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,1\,0 \end{array}$

3. $\begin{array}{rl} & 0\,0\,0\,0\,0\,1\,0\,1 \\ \text{Reading } \texttt{n} & 0\,0\,1\,0\,0\,1\,1\,0 \\ \hline D = & 0\,0\,0\,0\,0\,1\,0\,0 \end{array}$

4. $\begin{array}{rl} & 0\,0\,0\,0\,1\,0\,0\,1 \\ \text{Reading } \texttt{u} & 0\,0\,0\,1\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,0 \end{array}$

5. $\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \text{Reading } \texttt{a} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,1 \end{array}$

6. $\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,1\,1 \\ \text{Reading } \texttt{l} & 0\,0\,0\,0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,0 \end{array}$

7. $\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \text{Reading } \texttt{\_} & 0\,0\,0\,0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,0 \end{array}$

8. $\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \text{Reading } \texttt{a} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,1 \end{array}$

9. $\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,1\,1 \\ \text{Reading } \texttt{n} & 0\,0\,1\,0\,0\,1\,1\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,1\,0 \end{array}$

10. $\begin{array}{rl} & 0\,0\,0\,0\,0\,1\,0\,1 \\ \text{Reading } \texttt{n} & 0\,0\,1\,0\,0\,1\,1\,0 \\ \hline D = & 0\,0\,0\,0\,0\,1\,0\,0 \end{array}$

11. $\begin{array}{rl} & 0\,0\,0\,0\,1\,0\,0\,1 \\ \text{Reading } \texttt{o} & 1\,0\,0\,0\,1\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,1\,0\,0\,0 \end{array}$

12. $\begin{array}{rl} & 0\,0\,0\,1\,0\,0\,0\,1 \\ \text{Reading } \texttt{u} & 0\,0\,0\,1\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,1\,0\,0\,0\,0 \end{array}$

13. $\begin{array}{rl} & 0\,0\,1\,0\,0\,0\,0\,1 \\ \text{Reading } \texttt{n} & 0\,0\,1\,0\,0\,1\,1\,0 \\ \hline D = & 0\,0\,1\,0\,0\,0\,0\,0 \end{array}$

14. $$\begin{array}{ll} & 0\,1\,0\,0\,0\,0\,0\,1 \\ \text{Reading } \mathtt{c} & 0\,1\,0\,0\,0\,0\,0\,0 \\ \hline D = & 0\,1\,0\,0\,0\,0\,0\,0 \end{array}$$

15. $$\begin{array}{ll} & 1\,0\,0\,0\,0\,0\,0\,1 \\ \text{Reading } \mathtt{e} & 1\,0\,0\,0\,0\,0\,0\,0 \\ \hline D = & 1\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

The last bit is set; we mark an occurrence.

Example using DNA We search for the string **ATATA** in the sequence **AGATACGATATATAC**.

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

$D = 0\,0\,0\,0\,0$

1. $$\begin{array}{ll} & 0\,0\,0\,0\,1 \\ \text{Reading } \mathtt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 0\,0\,0\,0\,1 \end{array}$$

2. $$\begin{array}{ll} & 0\,0\,0\,1\,1 \\ \text{Reading } \mathtt{G} & 0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0 \end{array}$$

3. $$\begin{array}{ll} & 0\,0\,0\,0\,1 \\ \text{Reading } \mathtt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 0\,0\,0\,0\,1 \end{array}$$

4. $$\begin{array}{ll} & 0\,0\,0\,1\,1 \\ \text{Reading } \mathtt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,0\,0\,1\,0 \end{array}$$

5. $$\begin{array}{ll} & 0\,0\,1\,0\,1 \\ \text{Reading } \mathtt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 0\,0\,1\,0\,1 \end{array}$$

6. $$\begin{array}{ll} & 0\,1\,0\,1\,1 \\ \text{Reading } \mathtt{C} & 0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0 \end{array}$$

7. $$\begin{array}{ll} & 0\,0\,0\,0\,1 \\ \text{Reading } \mathtt{G} & 0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0 \end{array}$$

8. $$\begin{array}{ll} & 0\,0\,0\,0\,1 \\ \text{Reading } \mathtt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 0\,0\,0\,0\,1 \end{array}$$

9. $$\begin{array}{ll} & 0\,0\,0\,1\,1 \\ \text{Reading } \mathtt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,0\,0\,1\,0 \end{array}$$

10. $$\begin{array}{ll} & 0\,0\,1\,0\,1 \\ \text{Reading } \mathtt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 0\,0\,1\,0\,1 \end{array}$$

11. $$\begin{array}{ll} & 0\,1\,0\,1\,1 \\ \text{Reading } \mathtt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,1\,0\,1\,0 \end{array}$$

12. $$\begin{array}{ll} & 1\,0\,1\,0\,1 \\ \text{Reading } \mathtt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,1 \end{array}$$

The last bit is set; we mark an occurrence.

13. $$\begin{array}{ll} & 0\,1\,0\,1\,1 \\ \text{Reading } \mathtt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,1\,0\,1\,0 \end{array}$$

14. $$\begin{array}{ll} & 1\,0\,1\,0\,1 \\ \text{Reading } \mathtt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,1 \end{array}$$

The last bit is set; we mark an occurrence.

15. $$\begin{array}{ll} & 0\,1\,0\,1\,1 \\ \text{Reading } \mathtt{C} & 0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0 \end{array}$$

2.3 Suffix based approach

The main difficulty in the suffix based approach is to shift the window in a safe way, which means without missing an occurrence of the pattern.

We present the **Boyer-Moore** (**BM**) algorithm [BM77] and then the **Horspool** simplification [Hor80]. We do not give any pseudo-code for the first, nor a deeper study, for although **BM** improves over the algorithms of the other two general approaches, it is never the fastest.

2.3.1 Boyer-Moore idea

The **Boyer-Moore** algorithm precomputes three shift functions d_1, d_2, d_3 that correspond to the following three situations. For all of them, we have read a suffix u of the search window that is also a suffix of the pattern, and we have failed on a text character σ that does not match the next pattern character α (Figure 2.3).

First case The suffix u occurs in another position as a factor of p. Then a safe shift is to move the window so that u in the text matches the next occurrence of u in the pattern. This situation is shown in Figure 2.8. The idea is to compute for each suffix of the pattern the distance to the position of its next occurrence backwards in the pattern. We call this function d_1. If the suffix u of p does not appear again in p, then u is associated by d_1 to the size m of the whole pattern.

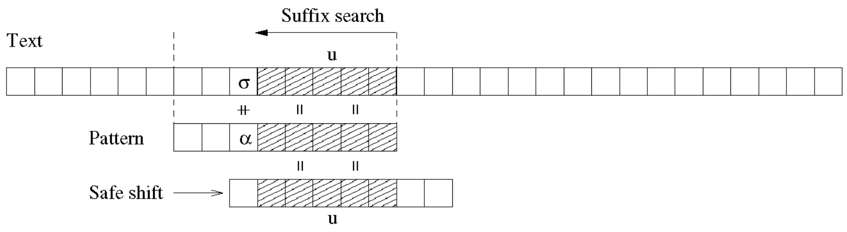

Fig. 2.8. First shift function d_1 of the **Boyer-Moore** algorithm. The pattern is shifted to the next occurrence of u.

Second case The suffix u does not occur in any other position as a factor of p. This does not mean that we can safely skip the whole search window, for the situation shown in Figure 2.9 can occur. A suffix v of u can also be a prefix of the pattern. To manage this case, we compute a second function d_2 for all suffixes of the pattern. It associates to each suffix u of p the length of the longest prefix v of p that is also a suffix of u.

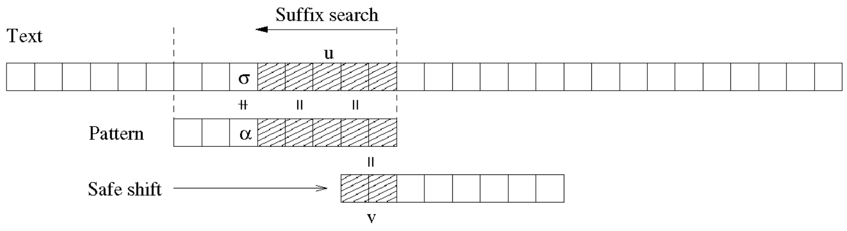

Fig. 2.9. Second shift function d_2 of the **Boyer-Moore** algorithm. No other occurrence of u exists in p. The pattern is shifted to the longest prefix of p that is also a suffix of u.

Third case The backward search has failed on the text character σ. If we shift the window with the first function d_1 and this letter is not aligned with a σ in the pattern, we will perform an unnecessary verification of the new search window. This case is shown in Figure 2.10. The third function, d_3, is computed to ensure that the text character σ will correspond to a σ in the pattern for the next verification. It associates to each character σ of the alphabet the distance of its rightmost occurrence to the end of the pattern. If a character σ does not occur in p, it is associated with m.

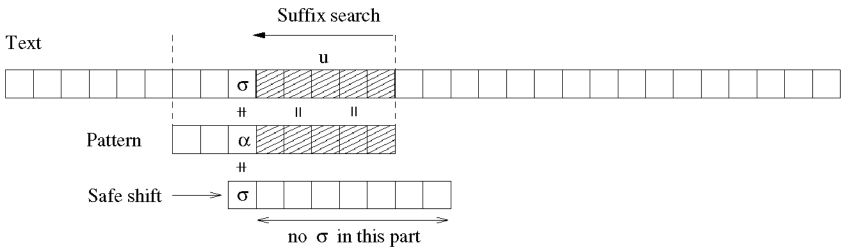

Fig. 2.10. Third shift function d_3 of the **Boyer-Moore** algorithm. The pattern is shifted to the next occurrence of σ in p.

To shift the window after we read u and failed on σ, the **Boyer-Moore** algorithm compares two shifts:

- the maximum between the shifts given by $d_1(u)$ and $d_3(\sigma)$, since we want to align u with its next occurrence in the pattern, knowing that the σ of the text has to match another σ in the pattern;
- the minimum between the result of the previous maximum and $m - d_2(u)$, since the latter expression is the maximum safe shift that can be performed.

However, if the beginning of the window has been reached, which means that we have found an occurrence, only the function d_2 is used to shift the search window.

The search part of **BM** has $O(mn)$ worst-case complexity, but it is sublinear on average. Many variations have been designed to make it linear in the worst case. The most important references are given in Section 2.6.

The main inconvenience of **BM** is the computation of the functions d_1, d_2, and d_3. They can be computed in $O(m)$ time, but that is difficult [Ryt80]. We now present a simplification that leads to algorithms that are more efficient than **BM** itself in numerous cases.

2.3.2 Horspool algorithm

The **BM** algorithm was first simplified by Horspool [Hor80], who assumed that, for a reasonably large alphabet, the shift function d_3 will always yield the longest shift. Horspool just considered a small modification of d_3 that is easy to compute and yields longer shifts. The resulting algorithm works as follows (Figure 2.11).

For each position of the search window, we compare its last character (β in the figure) with the last character of the pattern. If they match, we verify the search window backward against the pattern until we either find the pattern or fail on a text character (σ in the figure). Then, whether there was a match or not, we shift the window according to the next occurrence of the letter β in the pattern. Pseudo-code for the **Horspool** algorithm is given in Figure 2.12.

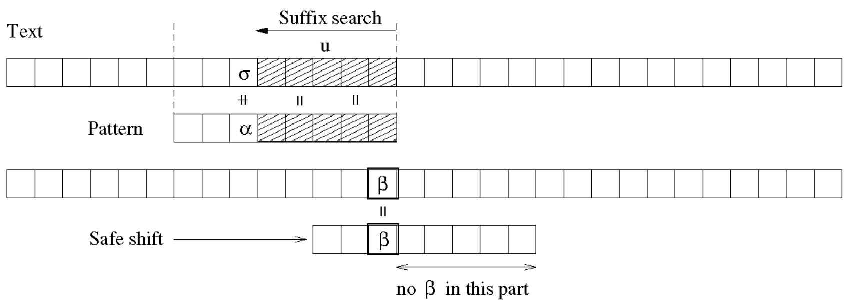

Fig. 2.11. **Horspool** algorithm. The pattern is shifted according to the last character of the search window.

```
Horspool ($p = p_1 p_2 \ldots p_m$, $T = t_1 t_2 \ldots t_n$)
1.   Preprocessing
2.       For $c \in \Sigma$ Do $d[c] \leftarrow m$
3.       For $j \in 1 \ldots m-1$ Do $d[p_j] \leftarrow m-j$
4.   Searching
5.       $pos \leftarrow 0$
6.       While $pos \leq n-m$ Do
7.           $j \leftarrow m$
8.           While $j > 0$ AND $t_{pos+j} = p_j$ Do $j \leftarrow j-1$
9.           If $j = 0$ Then report an occurrence at $pos+1$
10.          $pos \leftarrow pos + d[t_{pos+m}]$
11.      End of while
```

Fig. 2.12. **Horspool** algorithm.

We notice that:

- The verification also could have been done forward. Many implementations use a built-in memory comparison instruction.
- The main loop can be "unrolled," which means that we can first shift the search window until its last character matches the last character of the pattern, and then perform the verification.

The variant of Sunday Instead of shifting the window using its last character, we may use the next character after the window, which leads on average to longer shifts. This algorithm has been proposed by Sunday [Sun90]. Although the shifts are longer, the lower number of memory references of the *unrolled* **Horspool** algorithm makes it faster in general.

Example of the Horspool algorithm using English We search for the string "`announce`" in the text "`CPM_annual_conference_announce`".

$m = 8,\ d = \left\{ \begin{array}{|c|c|c|c|c|c|} \hline \texttt{a} & \texttt{c} & \texttt{n} & \texttt{o} & \texttt{u} & * \\ \hline 7 & 1 & 2 & 4 & 3 & 8 \\ \hline \end{array} \right.$

1. $\boxed{\texttt{CPM\_annu}}$ `al_conference_announce`

   $\texttt{u} \neq \texttt{e}$, $d[\texttt{u}] = 3$

2. `CPM` $\boxed{\texttt{\_annual\_}}$ `conference_announce`

   $\texttt{\_} \neq \texttt{e}$, $d[\texttt{\_}] = 8$

3. `CPM_annual_` $\boxed{\texttt{conferen}}$ `ce_announce`

   $\texttt{n} \neq \texttt{e}$, $d[\texttt{n}] = 2$

4. `CPM_annual_co` $\boxed{\texttt{nference}}$ `_announce`

   The last character [`nferenc`$\boxed{\texttt{e}}$] of the window matches the last character of the pattern. We continue the backward verification [`nferen`$\boxed{\texttt{ce}}$] , [`nfere`$\boxed{\texttt{nce}}$] , and it fails on the next character. We re-use the last character of the window, $d[\texttt{e}] = 8$.

5. `CPM_annual_conference` $\boxed{\texttt{\_announc}}$ `e`

   $\texttt{c} \neq \texttt{e}$, $d[\texttt{c}] = 1$

6. `CPM_annual_conference_` $\boxed{\texttt{announce}}$

   The last character [`announc`$\boxed{\texttt{e}}$] of the window matches the last character of the pattern. We verify backward the window and find the occurrence.

Example of the Horspool algorithm using DNA We search for the string `ATATA` in the sequence `AGATACGATATATAC`.

$m = 8,\ d = \left\{ \begin{array}{|c|c|c|} \hline \texttt{A} & \texttt{T} & * \\ \hline 2 & 1 & 5 \\ \hline \end{array} \right.$

1. $\boxed{\texttt{AGATA}}$ `CGATATATAC`

   The last character [`AGAT`$\boxed{\texttt{A}}$] of the window matches the last character of the pattern. We continue the backward verification [`AGA`$\boxed{\texttt{TA}}$] , [`AG`$\boxed{\texttt{ATA}}$] , and it fails on the next chararacter. We re-use the last character of the window, $d[\texttt{A}] = 2$.

2. `AG` $\boxed{\texttt{ATACG}}$ `ATATATAC`

   $\texttt{G} \neq \texttt{A}$, $d[\texttt{G}] = 5$

3. `AGATACG` $\boxed{\texttt{ATATA}}$ `TAC`

   The last character `[ ATAT`$\boxed{\texttt{A}}$` ]` of the window matches the last character of the pattern. We verify backward the window and find the occurrence. We then shift by re-using the last character of the window, $d[\texttt{A}] = 2$.

4. `AGATACGAT` $\boxed{\texttt{ATATA}}$ `C`

   The last character `[ ATAT`$\boxed{\texttt{A}}$` ]` of the window matches the last character of the pattern. We verify backward the window and find the new occurrence. We then shift by re-using the last character of the window, $d[\texttt{A}] = 2$. Then, $pos > n - m$ and the search stops.

2.4 Factor based approach

The factor based approach leads to optimal average-case algorithms, assuming that the characters of the text are independent and occur with the same probability.

The idea for moving the search window with this approach is elegant and simple. It is shown in Figure 2.13. Suppose that we have read backward a factor u of the pattern, and that we failed on the next letter σ. This means that the string σu is no longer a factor of p, so no occurrence of p can contain σu, and we can safely shift the window to after σ.

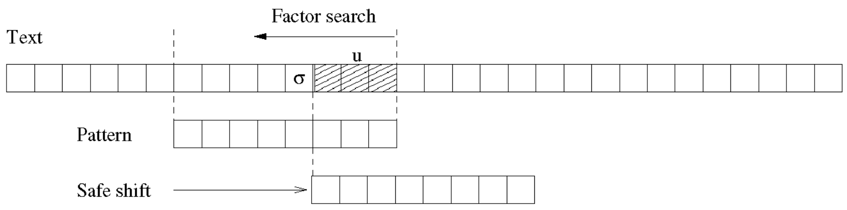

Fig. 2.13. Basic idea for shifting the window with the factor search approach. If we failed to recognize a factor of the pattern on σ, then σu is not a factor of the pattern and the window can be safely shifted after σ.

The main drawback to this approach is that it requires recognizing the set of factors of the pattern. We first present the **Backward Dawg Matching** (**BDM**) algorithm [CCG$^+$94]. This algorithm uses a suffix automaton, which is a powerful but complex structure. We will not describe it in this chapter for two reasons:

(i) When the pattern is short enough, of size less than w, the suffix automaton can be simulated efficiently with bit-parallelism. This algorithm, **Backward Nondeterministic Dawg Matching** [NR00], is faster than **BDM**, simpler to implement, and applicable to extended patterns (Chapter 4).

(ii) When the pattern is longer, the **Backward Oracle Matching** algorithm [ACR01], based on a modification of the factor based approach, leads to the same experimental times as **BDM**, but with a much simpler automaton, called the *factor oracle*.

2.4.1 Backward Dawg Matching idea

The **Backward Dawg Matching** algorithm uses a suffix automaton to perform the factor search, and it also improves the basic search approach. We begin with a general description of the *suffix automaton* and then explain the main parts of the algorithm.

Suffix automaton We need to recognize whether a given word u is a factor of the pattern p. There exist many indexing structures that enable us to determine whether u is a factor of p in $O(|u|)$ time. The most classic structure is the *compact suffix tree* [McC76]. However, in this structure, the transitions are coded as factors of the pattern, and to pass through a transition we need access to an arbitrary part of the pattern. The *suffix automaton* has the same efficiency, but its transitions are labeled with single characters. This speeds up the search and the pattern matching algorithms that use it. The interested reader can find a complete survey of the suffix automaton in [CH97, CR94]. We simply recall its three basic properties:

Pr_1 It enables us to determine whether a string u is a factor of a string p in $O(|u|)$ time. A string u is a factor in the suffix automaton built on p if and only if there is a path labeled u beginning at the initial node.

Pr_2 It enables us to recognize the suffixes of the pattern on which it is built. If a path beginning at the initial node reaches a terminal state of the automaton built on p, it means that the label of this path is a suffix of p.

Pr_3 It can be built on $p = p_1p_2 \ldots p_m$ in $O(m)$ time with an *on-line* algorithm, which means that the characters p_j can be added one after another into the structure, updating at each step j the suffix automaton of the prefix $p_1 \ldots p_{j-1}$ to obtain that of $p_1 \ldots p_j$.

Search algorithm The **BDM** algorithm [CCG+94] makes use of the properties of the suffix automaton. The general approach of Figure 2.13 is possible using the suffix automaton. Moreover, property Pr_2 enables a tricky improvement.

To search a pattern $p = p_1p_2 \ldots p_m$ in a text $T = t_1t_2 \ldots t_n$, the suffix automaton of $p^{rv} = p_mp_{m-1} \ldots p_1$ is built. The algorithm searches backwards along the window for a factor of the pattern using the suffix automaton.

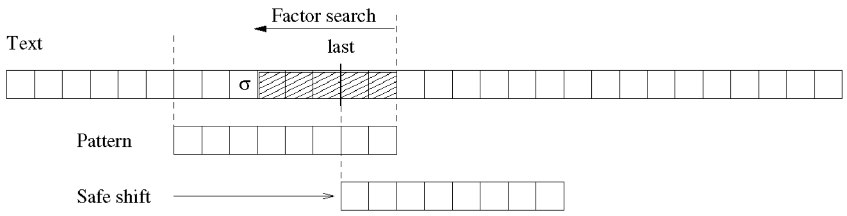

Fig. 2.14. Basic search of the **BDM** algorithm with the suffix automaton. The variable *last* stores the beginning position of the longest suffix of the part read that is also a prefix of the pattern.

During this search, if a terminal state is reached that does not correspond to the entire pattern, the position in the window is stored in a variable *last*. Due to property Pr_2, this corresponds to finding a *prefix* of the pattern starting at position *last* inside the window and ending at the end of the window since the suffixes of p^{rv} are the reverse prefixes of p. Since we stored the last prefix recognized backwards, we have the *longest* prefix of p in the window. This backward search ends in two possible ways:

(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}. We then shift the window so that its new starting position corresponds to the position *last*. We cannot miss an occurrence because in that case the suffix automaton would have found its prefix in the window. This situation is shown in Figure 2.14.

(ii) We reach the beginning of the window, thus recognizing the pattern p. We report the occurrence, and we shift the window exactly as in the previous case.

The algorithm is $O(mn)$ time in the worst case. However, it is the optimal $O(n \log_{|\Sigma|} m/m)$ on average under the assumption that the text characters are independent and have the same occurrence probabilities.

2.4.2 Backward Nondeterministic Dawg Matching algorithm

The **Backward Nondeterministic Dawg Matching** (**BNDM**) algorithm uses the same search approach as **BDM**, but the factor is searched using bit-parallelism. Compared to the original **BDM** algorithm, **BNDM** is simpler, uses less memory, has more locality of reference, and is easier to extend to more complex patterns (Chapter 4).

The idea is to maintain a set of positions on the reverse pattern that are the beginning positions of the string u read in the text. This set is stored with 0 and 1 as with **Shift-And**. The number 1, representing an active state at position j of p, means that the factor $p_j \dots p_{j+|u|-1}$ is equal to u. Figure 2.15 shows this relationship. If the pattern is of size less that w, then the set fits in a computer word $D = d_m \dots d_1$.

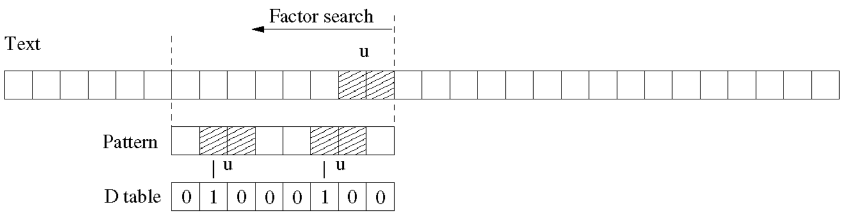

Fig. 2.15. Bit-parallel factor search. The table D keeps a list of the positions in p where the factor u begins.

We need to update the array D to D' after reading a new character σ of the text. A state j of D' is active if it corresponds to the beginning of the string σu in the pattern; that is, if

- u began at position $j+1$ in the pattern, which means that the $(j+1)$-th position in D is active, and
- σ is in position j in the pattern.

If we precompute a table B exactly as for **Shift-And** that associates to each letter of p the set of its positions in p with a bit mask, then we obtain D' from D by the following formula:

$$D' \;\leftarrow\; (D \;<<\; 1) \;\;\&\;\; B[\sigma] \tag{2.2}$$

However, there is a problem with the initialization. We would like to mark in the initial table D that each position of D matches the empty string, which means D should be 1^m. But in that case, the first shift will give $(D << 1) = 1^{m-1}0$ and we will miss the first factor, which corresponds to the entire word. The simplest solution would be to take D of size $m+1$, initialized to 1^{m+1}. However, it reduces to $w-1$ the maximum length of the string that can be searched. Instead we split formula (2.2) into two parts.

We first perform the operation $D'_1 \;\leftarrow\; D \;\&\; B[\sigma]$ and verify the match, and then we perform the register shift $D' \;\leftarrow\; D'_1 \;<<\; 1$. The initialization is then $D = 1^m$. A string read in the text is a prefix of p if the first position is active, that is, if in D'_1 the position d_m is active.

The **BNDM** algorithm is the same as **BDM**, except that the factor search is done with the bit-parallelism technique. Each time the bit d_m is active,

```
BNDM ($p = p_1p_2 \ldots p_m$, $T = t_1t_2 \ldots t_n$)
    Preprocessing
        For $c \in \Sigma$ Do $B[c] \leftarrow 0^m$
        For $j \in 1 \ldots m$ Do $B[p_j] \leftarrow B[p_j] \mid 0^{j-1}10^{m-j}$
    Searching
        $pos \leftarrow 0$
        While $pos \leq n - m$ Do
            $j \leftarrow m$, $last \leftarrow m$
            $D \leftarrow 1^m$
            While $D \neq 0^m$ Do
                $D \leftarrow D$ & $B[t_{pos+j}]$
                $j \leftarrow j - 1$
                If $D$ & $10^{m-1} \neq 0^m$ Then
                    If $j > 0$ Then $last \leftarrow j$
                    Else report an occurrence at $pos + 1$
                End of if
                $D \leftarrow D << 1$
            End of while
            $pos \leftarrow pos + last$
        End of while
```

Fig. 2.16. Bit-parallel pseudo-code for **BNDM**.

the position of the window is stored in the variable *last*. Pseudo-code for the algorithm is given in Figure 2.16.

BNDM has the same worst-case complexity $O(mn)$ as **BDM**, and also the same optimal average complexity $O(n \log_{|\Sigma|} m/m)$.

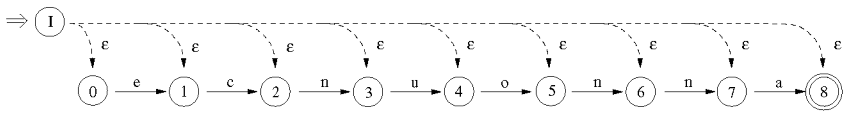

Fig. 2.17. Nondeterministic automaton recognizing all factors of the reverse string of "announce".

From an automaton point of view, the bit-parallel factor search is a simulation of a nondeterministic automaton that recognizes all suffixes of the reverse pattern. For example, if we search the pattern "announce", we simulate the automaton shown in Figure 2.17. It turns out that the minimal deterministic version of this automaton is the *suffix automaton* used in the classic **BDM**. The difference between **BNDM** and **BDM** is conceptually the same as that between **Shift-Or** and **KMP**. The former simulates a nondeterministic automaton using bit-parallelism, and the latter first obtains a representation of the deterministic automaton.

Example of BNDM using English We search for the string "announce" in the text "CPM_annual_conference_announce".

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

$D = 1\,1\,1\,1\,1\,1\,1\,1$

1. $\boxed{\texttt{CPM\_annu}}\ \texttt{al\_conference\_announce}$

$last \leftarrow 8$

$$\begin{array}{rl} & 1\,1\,1\,1\,1\,1\,1\,1 \\ \text{Reading } \texttt{u} & 0\,0\,0\,0\,1\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,1\,0\,0\,0 \end{array}$$

$$\begin{array}{rl} & 0\,0\,0\,1\,0\,0\,0\,0 \\ \text{Reading } \texttt{n} & 0\,1\,1\,0\,0\,1\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

2. $\texttt{CPM\_annu}\ \boxed{\texttt{al\_confe}}\ \texttt{rence\_announce}$

$last \leftarrow 8$

$$\begin{array}{rl} & 1\,1\,1\,1\,1\,1\,1\,1 \\ \text{Reading } \texttt{e} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,1 \end{array}$$

$$\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,1\,0 \\ \text{Reading } \texttt{f} & 0\,0\,0\,0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

3. $\texttt{CPM\_annual\_confe}\ \boxed{\texttt{rence\_an}}\ \texttt{nounce}$

$last \leftarrow 8$

$$\begin{array}{rl} & 1\,1\,1\,1\,1\,1\,1\,1 \\ \text{Reading } \texttt{n} & 0\,1\,1\,0\,0\,1\,0\,0 \\ \hline D = & 0\,1\,1\,0\,0\,1\,0\,0 \end{array}$$

$$\begin{array}{rl} & 1\,1\,0\,0\,1\,0\,0\,0 \\ \text{Reading } \texttt{a} & 1\,0\,0\,0\,0\,0\,0\,0 \\ \hline D = & 1\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

The position d_8 is active, but $j > 0$, so we set $last \leftarrow 6$.

4. $\texttt{CPM\_annual\_conference\_}\ \boxed{\texttt{announce}}$

$last \leftarrow 8$

$$\begin{array}{rl} & 1\,1\,1\,1\,1\,1\,1\,1 \\ \text{Reading } \texttt{e} & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline D = & 0\,0\,0\,0\,0\,0\,0\,1 \end{array}$$

$$\begin{array}{rl} & 0\,0\,0\,0\,0\,0\,1\,0 \\ \text{Reading } \texttt{c} & 0\,0\,0\,0\,0\,0\,1\,0 \\ \hline D = & 0\,0\,0\,0\,0\,0\,1\,0 \end{array}$$

$$\begin{array}{rl} & 0\,0\,0\,0\,0\,1\,0\,0 \\ \text{Reading } \texttt{n} & 0\,1\,1\,0\,0\,1\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0\,1\,0\,0 \end{array}$$

$$\begin{array}{rl} & 0\,0\,0\,0\,1\,0\,0\,0 \\ \text{Reading } \texttt{u} & 0\,0\,0\,0\,1\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,1\,0\,0\,0 \end{array}$$

$$\begin{array}{rl} & 0\,0\,0\,1\,0\,0\,0\,0 \\ \text{Reading } \texttt{o} & 0\,0\,0\,1\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,1\,0\,0\,0\,0 \end{array}$$

$$\begin{array}{rl} & 0\,0\,1\,0\,0\,0\,0\,0 \\ \text{Reading } \texttt{n} & 0\,1\,1\,0\,0\,1\,0\,0 \\ \hline D = & 0\,0\,1\,0\,0\,0\,0\,0 \end{array}$$

$$\begin{array}{rl} & 0\,1\,0\,0\,0\,0\,0\,0 \\ \text{Reading } \texttt{n} & 0\,1\,1\,0\,0\,1\,0\,0 \\ \hline D = & 0\,1\,0\,0\,0\,0\,0\,0 \end{array}$$

$$\begin{array}{rl} & 1\,0\,0\,0\,0\,0\,0\,0 \\ \text{Reading } \texttt{a} & 1\,0\,0\,0\,0\,0\,0\,0 \\ \hline D = & 1\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

The position d_8 is active and $j = 0$, so we mark an occurrence.

Example of BNDM using DNA We search for the string `ATATA` in the sequence `AGATACGATATATAC`.

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

$D = 1\,1\,1\,1\,1$

1. $\boxed{\texttt{AGATA}}$ `CGATATATAC`

$last \leftarrow 5$

$$\begin{array}{rl} & 1\,1\,1\,1\,1 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,1 \end{array}$$

$$\begin{array}{rl} & 0\,1\,0\,1\,0 \\ \text{Reading } \texttt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,1\,0\,1\,0 \end{array}$$

$$\begin{array}{rl} & 1\,0\,1\,0\,0 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,0 \end{array}$$

The position d_5 is active, but $j > 0$, so we set $last \leftarrow 2$.

$$\begin{array}{rl} & 0\,1\,0\,0\,0 \\ \text{Reading } \texttt{G} & 0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0 \end{array}$$

2. `AG` $\boxed{\texttt{ATACG}}$ `ATATATAC`

$last \leftarrow 5$

$$\begin{array}{rl} & 1\,1\,1\,1\,1 \\ \text{Reading } \texttt{G} & 0\,0\,0\,0\,0 \\ \hline D = & 0\,0\,0\,0\,0 \end{array}$$

3. `AGATACG` $\boxed{\texttt{ATATA}}$ `TAC`

$last \leftarrow 5$

$$\begin{array}{rl} & 1\,1\,1\,1\,1 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,1 \end{array}$$

$$\begin{array}{rl} & 0\,1\,0\,1\,0 \\ \text{Reading } \texttt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,1\,0\,1\,0 \end{array}$$

$$\begin{array}{rl} & 1\,0\,1\,0\,0 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,0 \end{array}$$

The position d_5 is active, but $j > 0$, so we set $last \leftarrow 2$.

$$\begin{array}{rl} & 0\,1\,0\,0\,0 \\ \text{Reading } \texttt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,1\,0\,0\,0 \end{array}$$

$$\begin{array}{rl} & 1\,0\,0\,0\,0 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,0\,0\,0 \end{array}$$

The position d_5 is active and $j = 0$, so we mark an occurrence.

4. `AGATACGAT` $\boxed{\texttt{ATATA}}$ `C`

$last \leftarrow 5$

$$\begin{array}{rl} & 1\,1\,1\,1\,1 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,1 \end{array}$$

$$\begin{array}{rl} & 0\,1\,0\,1\,0 \\ \text{Reading } \texttt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,1\,0\,1\,0 \end{array}$$

$$\begin{array}{rl} & 1\,0\,1\,0\,0 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,1\,0\,0 \end{array}$$

The position d_5 is active, but $j > 0$, so we set $last \leftarrow 2$.

$$\begin{array}{rl} & 0\,1\,0\,0\,0 \\ \text{Reading } \texttt{T} & 0\,1\,0\,1\,0 \\ \hline D = & 0\,1\,0\,0\,0 \end{array}$$

$$\begin{array}{rl} & 1\,0\,0\,0\,0 \\ \text{Reading } \texttt{A} & 1\,0\,1\,0\,1 \\ \hline D = & 1\,0\,0\,0\,0 \end{array}$$

The position d_5 is active and $j = 0$, so we mark a new occurrence. We then shift to $pos + last$ and $pos > n - m$, so the search stops.

2.4.3 Backward Oracle Matching algorithm

For patterns longer than w, the normal **BDM** algorithm would be necessary but the complexity of the construction of the suffix automaton makes it impractical. A solution has been proposed recently [ACR01]. It is based on the observation that, to shift the window in the general factor search approach (Figure 2.13), it is not necessary to know that u is a factor. It suffices to know that σu is not.

The *factor oracle* structure has this particularity. Built on a string p, it recognizes *more* than the set of factors of p, but it is easy to understand and implement and is compact, so that the efficiency lost by reading more letters in the backward search is recovered by doing fewer page faults.

To simplify notation, we denote by θ an object that is not defined. For instance, in an automaton, $\delta(q, \alpha) = \theta$ means that there is no outgoing transition from q labeled with α.

2.4.3.1 Factor oracle

The *factor oracle* built on a string $p = p_1 p_2 \dots p_m$ is a deterministic acyclic automaton that has $m + 1$ states and m to $2m - 1$ transitions. We denote its transition function by δ.

The $m+1$ states correspond to the $m+1$ positions between the characters of p, including a first position 0 before the whole pattern. A state $0 < i \leq m$ corresponds to the prefix $p_1 \dots p_i$.

The first m transitions spell out the pattern itself in a line; for $0 < i \leq m$, we build a transition from state $i - 1$ to i labeled p_i. In practice, these transitions and states can be stored implicitly with the pattern itself.

Then, we build what we call the "external transitions," of which there are at most $m - 1$. We associate to each state i another state $j < i$, called its "supply state" and denoted $j = S(i)$. This function is the "supply function." It is built together with the external transitions. $S(0)$ is set to θ.

The construction algorithm proceeds by inspecting each state from 1 to m. We assume that we have reached state $i - 1$ and begun to inspect the i-th state. We go down the supply function from state $i - 1$. We use a variable k initialized to $S(i - 1)$ and we repeat the following steps.

ST_1 If $k = \theta$, then $S(i) \leftarrow 0$.

ST_2 If $k \neq \theta$ and there does not exist a transition from state k labeled by p_i, then build a transition from state k to state i by p_i, and return to step ST_1 with $k \leftarrow S(k)$.

ST_3 If $k \neq \theta$ and there exists a transition from k labeled by p_i leading to a state j, then set $S(i) \leftarrow j$ and stop processing state i.

This construction is simple. Moreover, it is clear that it can be done *on-line*, which means that we can add the letters p_i one after another and build the new state i and all the new transitions at this time. Pseudo-code for the *on-line* construction is given in Figure 2.18. The algorithm is linear in the size of the pattern.

```
Oracle_add_letter(Oracle($p = p_1p_2 \ldots p_m$), $\sigma$)
1.     Create a new state $m + 1$
2.     $\delta(m, \sigma) \leftarrow m + 1$
3.     $k \leftarrow S(m)$
4.     While $k \neq \theta$ AND $\delta(k, \sigma) = \theta$ Do
5.         $\delta(k, \sigma) \leftarrow m + 1$
6.         $k \leftarrow S(k)$
7.     End of while
8.     If $k = \theta$ Then $s \leftarrow 0$
9.     Else $s \leftarrow \delta(k, \sigma)$
10.    $S(m + 1) \leftarrow s$
11.    Return  Oracle($p = p_1p_2 \ldots p_m\sigma$)

Oracle-on-line($p = p_1p_2 \ldots p_m$)
12.    Create Oracle($\varepsilon$) with:
13.        One single initial state 0
14.        $S(0) \leftarrow \theta$
15.    For $j \in 1 \ldots m$ Do
16.        Oracle($p = p_1p_2 \ldots p_j$) $\leftarrow$ Oracle_add_letter(Oracle($p = p_1p_2 \ldots p_{j-1}$), $p_j$)
17.    End of for
```

Fig. 2.18. Construction of the factor oracle. The function **Oracle_add_letter** adds a letter σ to Oracle($p = p_1p_2 \ldots p_m$) to get Oracle($p\sigma$). The *on-line* construction algorithm adds the letters p_i one by one to obtain finally Oracle($p = p_1p_2 \ldots p_m$).

The factor oracle built on p recognizes all the factors of p. It really recognizes more, but not so many in practice, and it recognizes only one string of size m, the pattern itself.

To code it, the easiest way in practice is to use a $(m+1) \times A$ table, where A is the alphabet size of the pattern. This representation has the advantage of giving $O(1)$ access time to the transitions, which speeds up the search algorithm. However, for very long patterns, an implementation in $O(m)$ space has to be considered.

2.4.3.2 Search with the factor oracle

The search algorithm with the factor oracle, called **Backward Oracle Matching (BOM)**, is the simple transcription of the factor search approach (Figure 2.13). We read backwards in the window the text characters in the factor oracle of the reverse pattern p^{rv}. If we fail on a letter σ after reading a string u, we know that σu is not a factor of p and we can safely shift the window after the letter σ. If the beginning of the window is reached, then, since the factor oracle recognizes only one string of size $|p|$, we mark a match and we shift the window by one character. Pseudo-code for **BOM** is given in Figure 2.19.

```
BOM($p = p_1p_2\dots p_m$, $T = t_1t_2\dots t_n$)
1.     Preprocessing
2.         Oracle-on-line($p^{rv}$)
               $\delta$ is its transition function
3.     Searching
4.         $pos \leftarrow 0$
5.         While $pos \le n - m$ Do
6.             $Current \leftarrow$ initial state of Oracle($p^{rv}$)
7.             $j \leftarrow m$
8.             While $j > 0$ AND $Current \ne \theta$ Do
9.                 $Current \leftarrow \delta(Current, t_{pos+j})$
10.                $j \leftarrow j - 1$
11.            End of while
12.            If $Current \ne \theta$ Then
13.                mark an occurrence at $pos + 1$
14.            End of if
15.            $pos \leftarrow pos + j + 1$
16.        End of while
```

Fig. 2.19. Pseudo-code of the **BOM** algorithm.

BOM is $O(mn)$ time in the worst case. From experimental results it is conjectured that it is optimal on average.

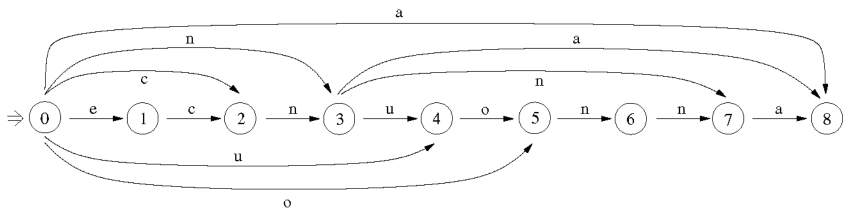

Fig. 2.20. Factor oracle for the reverse string of “announce”.

Example using English We search for the string "announce" in the text "CPM_annual_conference_announce". The factor oracle of the reverse pattern of "announce" is given in Figure 2.20.

1. $\boxed{\texttt{CPM\_annu}}$ al_conference_announce

   Reading [CPM_ann$\boxed{\texttt{u}}$] in the factor oracle.

   Fail on the next character n.

2. CPM_ann $\boxed{\texttt{ual\_conf}}$ erence_announce

   Fail on the character f.

3. CPM_annual_conf $\boxed{\texttt{erence\_a}}$ nnounce

   Reading [erence_$\boxed{\texttt{a}}$] in the factor oracle.

   Fail on the next character _.

4. CPM_annual_conference_ $\boxed{\texttt{announce}}$

   Reading [announc$\boxed{\texttt{e}}$] in the factor oracle.

   Reading [announ$\boxed{\texttt{c}}$e] in the factor oracle.

   Reading [annou$\boxed{\texttt{n}}$ce] in the factor oracle.

   Reading [anno$\boxed{\texttt{u}}$nce] in the factor oracle.

   Reading [ann$\boxed{\texttt{o}}$unce] in the factor oracle.

   Reading [an$\boxed{\texttt{n}}$ounce] in the factor oracle.

   Reading [a$\boxed{\texttt{n}}$nounce] in the factor oracle.

   Reading [$\boxed{\texttt{a}}$nnounce] in the factor oracle.

   We mark an occurrence.

Example using DNA We search for the string ATATA in the sequence AGATACGATATATAC. The factor oracle of the reverse pattern of ATATA is given in Figure 2.21.

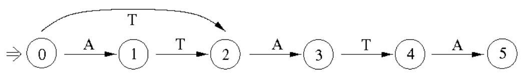

Fig. 2.21. Factor oracle for the reverse string of ATATA.

1. $\boxed{\texttt{AGATA}}$ CGATATATAC

   Reading [AGAT$\boxed{\texttt{A}}$] in the factor oracle.

   Reading [AGA$\boxed{\texttt{T}}$A] in the factor oracle.

   Reading [AG$\boxed{\texttt{A}}$TA] in the factor oracle.

   Fail on the next character G.

2. AG $\boxed{\texttt{ATACG}}$ ATATATAC

   Fail on the character G.

3. AGATACG $\boxed{\texttt{ATATA}}$ TAC

   Reading [ATAT$\boxed{\texttt{A}}$] in the factor oracle.

   Reading [ATA$\boxed{\texttt{T}}$A] in the factor oracle.

   Reading [AT$\boxed{\texttt{A}}$TA] in the factor oracle.

   Reading [A$\boxed{\texttt{T}}$ATA] in the factor oracle.

   Reading [$\boxed{\texttt{A}}$TATA] in the factor oracle.

   We mark an occurrence.

4. AGATACGA $\boxed{\texttt{TATAT}}$ AC

   Reading [TATA$\boxed{\texttt{T}}$] in the factor oracle.

   Reading [TAT$\boxed{\texttt{A}}$T] in the factor oracle.

   Reading [TA$\boxed{\texttt{T}}$AT] in the factor oracle.

   Reading [T$\boxed{\texttt{A}}$TAT] in the factor oracle.

   Fail on the character T.

5. AGATACGAT $\boxed{\texttt{ATATA}}$ C

   Reading [ATAT$\boxed{\texttt{A}}$] in the factor oracle.

   Reading [ATA$\boxed{\texttt{T}}$A] in the factor oracle.

   Reading [AT$\boxed{\texttt{A}}$TA] in the factor oracle.

   Reading [A$\boxed{\texttt{T}}$ATA] in the factor oracle.

   Reading [$\boxed{\texttt{A}}$TATA] in the factor oracle.

   We mark a new occurrence.

6. AGATACGATA $\boxed{\texttt{TATAC}}$

   Fail on the character C.

2.5 Experimental map

We present in this section a map of the efficiency of different string matching algorithms, showing zones where they are most efficient in practice. The experiments were performed on a $w = 32$ bits Ultra Sparc 1 running SunOs 5.6. Texts of 10 megabytes were randomly built, as were the patterns. The experiments were repeated until we obtained a relative error below 2% with 95% confidence. We tested optimized implementations of all the algorithms presented. However, only **Shift-Or**, **Horspool**, **BNDM**, and **BOM** have a zone in the map, since the others were too slow.

The map is shown in Figure 2.22. We show the length w of a register word to recall that it is the maximum size of string that **BNDM** can manage with a single word implementation.

Results on DNA sequences turn out to be the same as those for a random text of size 4. A more surprising fact is that results on English are about the same as those for a random text of size 16.

The map shows clearly that the **Horspool** algorithm becomes more and more difficult to beat as the alphabet grows. The **BNDM** algorithm is confined to a small zone for small alphabet sizes, but the map does not reflect its ability to handle extended strings. The **Shift-Or** wins only for small strings on very small alphabet sizes.

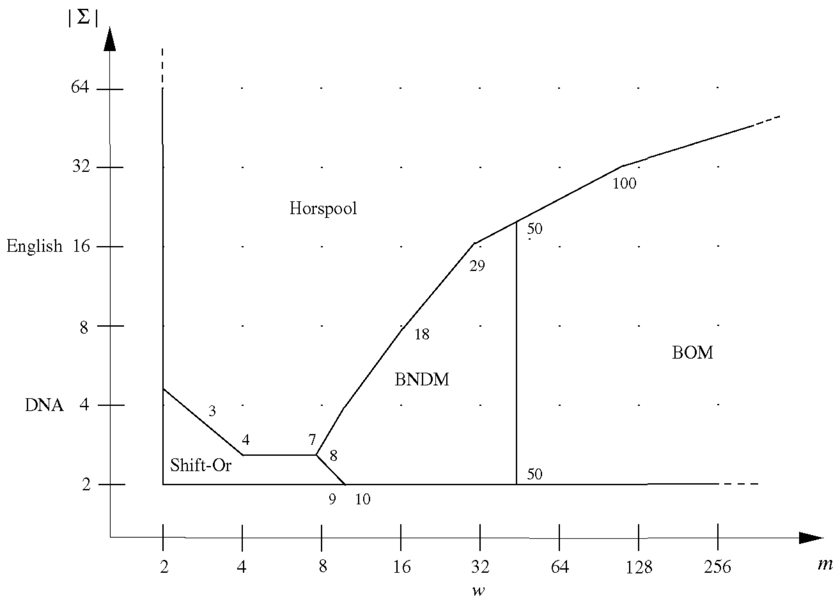

Fig. 2.22. Map of experimental efficiency for different string matching algorithms.

2.6 Other algorithms and references

Many other algorithms exist for searching a string in a text. We give in this section the most important references on string matching research.

On the Knuth-Morris-Pratt algorithm Many variants exist based on **MP** [MP70] and **KMP** [KMP77], the most important one being the **Simon** algorithm [Sim93]. Simon shows that the underlying automaton of **KMP** can be completed and stored in an efficient way. Some complete analyses on **KMP** can be found in [Rég89]. The **Simon** algorithm has been analyzed in [Han93].

On the Boyer-Moore algorithm As for **KMP**, many variants of **BM** [BM77] exist. The principal ones are the **Boyer-Moore-Galil** [Gal79] and the **Turbo-BM** [CGR92] algorithms. The **BM** algorithm has been analyzed in [BYGR90, BYR92, Col94]. The underlying automaton was analyzed in [BYG89a, Cho90, BYCG94, BBYDS96]. The **Horspool** algorithm has been analyzed in [MRS96].

On the Backward Dawg Matching algorithm The **BDM** algorithm used together with a **KMP** algorithm is linear in the worst case. An example of this is the **TurboBDM** algorithm [CCG$^+$94], and another is **TurboRF** [CCG$^+$94]. The **Double Forward Dawg Matching** algorithm [AR00] is the simplest worst-case linear time and optimal on average.

Constant space algorithms In 1981 there appeared in [GS81] the first linear time string matching algorithm that uses only a constant amount of additional space. Since then, many others have appeared [CP91, Cro92, CR95]. Finding a constant space algorithm that is optimal on average is an open problem.

Hashing The most famous hashing algorithm is **Karp-Rabin** [KR87]. It is analyzed in [GBY90].