

Regular expression matching

5.1 Basic concepts

We present in this chapter algorithms to search for regular expressions in texts or biological sequences. Regular expressions are often used in text retrieval or computational biology applications to represent search patterns that are more complex than a string, a set of strings, or an extended string. We begin with a formal definition of a regular expression and the language (set of strings) it represents.

Definition *A regular expression RE is a string on the set of symbols $\Sigma \cup \{ \varepsilon, |, \cdot, *, (,) \}$, which is recursively defined as the empty character ε ; a character $\alpha \in \Sigma$; and (RE_1) , $(RE_1 \cdot RE_2)$, $(RE_1 | RE_2)$, and (RE_1^*) , where RE_1 and RE_2 are regular expressions.*

For instance, in this chapter we consider the regular expression $((A \cdot T) | (G \cdot A)) \cdot (((A \cdot G) | ((A \cdot A) \cdot A))^*)$. When there is no ambiguity, we simplify our expressions by writing $RE_1 RE_2$ instead of $(RE_1 \cdot RE_2)$. This way, we obtain a more readable expression, in our case $(AT|GA)((AG|AAA)^*)$. It is usual to use also the precedence order “ $*$ ”, “ \cdot ”, “ $|$ ” to remove more parentheses, but we do not do this here. The symbols “ \cdot ”, “ $|$ ”, “ $*$ ” are called *operators*. It is customary to add an extra postfix operator “ $+$ ” to mean $RE^+ = RE \cdot RE^*$. We define now the language represented by a regular expression.

Definition *The language represented by a regular expression RE is a set of strings over Σ , which is defined recursively on the structure of RE as follows:*

- If RE is ε , then $L(RE) = \{\varepsilon\}$, the empty string.
- If RE is $\alpha \in \Sigma$, then $L(RE) = \{\alpha\}$, a single string of one character.
- If RE is of the form (RE_1) , then $L(RE) = L(RE_1)$.

- If RE is of the form $(RE_1 \cdot RE_2)$, then $L(RE) = L(RE_1) \cdot L(RE_2)$, where $W_1 \cdot W_2$ is the set of strings w such that $w = w_1w_2$, with $w_1 \in W_1$ and $w_2 \in W_2$. The operator “ \cdot ” represents the classical concatenation of strings.
- If RE is of the form $(RE_1 \mid RE_2)$, then $L(RE) = L(RE_1) \cup L(RE_2)$, the union of the two languages. We call the symbol “ \mid ” the union operator.
- If RE is (RE_1^*) , then $L(RE) = L(RE_1)^* = \bigcup_{i \geq 0} L(RE_1)^i$, where $L^0 = \{\varepsilon\}$ and $L^i = L \cdot L^{i-1}$ for any L . That is, the result is the set of strings formed by a concatenation of zero or more strings represented by RE_1 . We call “ $*$ ” the star operator.

For instance, $L((AT|GA)((AG|AAA)^*)) = \{ AT, GA, ATAG, GAAG, ATA- AA, GAAAA, ATAGAG, ATAGAAA, ATAAAAG, ATAAAAAA, GAAGAG, GAAGAAA, \dots \}$. Note that, according to the definition of the star operator, Σ^* denotes the set of all the strings over the alphabet Σ .

The *size* of a regular expression RE is the number of characters of Σ inside it. For instance, the size of $(AT|GA)((AG|AAA)^*)$ is 9. The complexities of the algorithms that we present below are based on this measure.

The problem of searching for a regular expression RE in a text T is to find all the factors of T that belong to the language $L(RE)$. We present in this chapter the main strategies for performing this search.

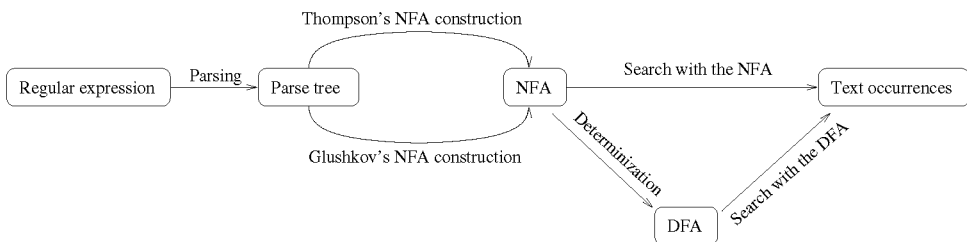


Fig. 5.1. The classical approaches to searching for regular expressions in a text.

Figure 5.1 summarizes the classical approaches. The regular expression is first parsed into an expression tree, which is transformed into a Nondeterministic Finite Automaton (NFA) in several possible ways. In this chapter we first present two NFA constructions, which are the most interesting in practice. The first one is the Thompson construction [Tho68], and the second is the Glushkov construction [Glu61].

It is possible to search directly with the NFA, and there are various ways to do that, but the process is quite slow. The algorithm consists in keeping

a list of active states and updating the list each time a new text character is read. The search is normally worst-case time $O(mn)$, but it requires little memory.

Another approach is to convert the NFA into a Deterministic Finite Automaton (DFA), which permits $O(n)$ search time by performing one direct transition per text character. On the other hand, the construction of such an automaton is worst-case time and space $O(2^m)$.

Yet a third strategy is to filter the text using multiple pattern matching or related tools, so as to find anchors around which there might be an occurrence, and then locally verify a possible occurrence using one of the previous strategies. Figure 5.2 illustrates this scheme.



Fig. 5.2. The filtering approach to search for regular expressions in a text.

These strategies can be combined. Moreover, the use of bit-parallelism can accelerate some parts of the search process.

An important point is that most of the automaton constructions use a tree representation of the regular expression RE in order to perform the calculations. The leaves of the tree are labeled with the characters of Σ in RE and also with the symbols ε , if any. The internal nodes are labeled with the operators. The nodes that are labeled “|” or “.” have two children that represent the subexpressions RE_1 and RE_2 (Section 5.1). Nodes labeled “*” have a unique child representing RE_1 . The tree representation is usually not unique, since some operators are commutative and/or associative. A tree representation of our example $(AT|GA)((AG|AAA)*)$ is shown in Figure 5.3.

We explain in Section 5.8 how to parse a regular expression in order to obtain a tree representation. We consider below that the parse tree is readily available and identify our regular expressions with any of their tree representations.

When working on the tree representations in our algorithms, we assume that the symbol $\boxed{\cdot}(v_l, v_r)$ means a concatenation tree with root “.” and children v_l and v_r . Similarly, $\boxed{|}(v_l, v_r)$ is the tree rooted with “|”. The symbol $\boxed{*}(v_*)$ means a “*” node with a unique child v_* .

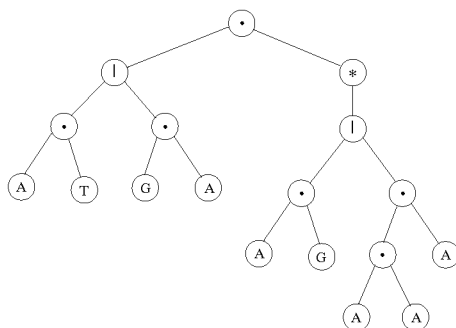


Fig. 5.3. Tree representation of the regular expression $(AT|GA)((AG|AAA)^*)$.

5.2 Building an NFA

There exist various ways to build an NFA from a regular expression [Glu61, Tho68, CP92, BS86, BK93, HSW97], among which two are most important because they are practical and often used.

The Thompson construction [Tho68] is simple and leads to an NFA that is linear in the number of states (at most $2m$) and of transitions (at most $4m$). However, this automaton has ε -transitions, that is, “empty” transitions, that can be passed through without reading a character of the text or, alternatively, by reading the empty string ε .

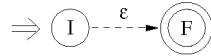
The Glushkov construction [Glu61, BS86], on the other hand, leads to an NFA with exactly $m + 1$ states but a number of transitions that is $O(m^2)$ in the worst case. Nevertheless, this construction produces no ε -transitions. The original construction is $O(m^3)$ time, but it has been shown [BK93] that this can be reduced to $O(m^2)$.

5.2.1 Thompson automaton

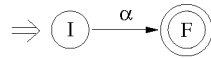
The construction of Thompson [Tho68] is an automaton representation of what is recognized by the regular expression. The automaton is a direct transcription of the tree representation of the regular expression. It uses ε -transitions to simplify this transcription.

The idea is to go up the tree representation T_{RE} of the regular expression RE and to compute for each tree node v an automaton $Th(v)$ that recognizes the language RE_v represented by the subtree rooted at v . A specific automaton construction is associated to each type of node and leaf of the tree. These are

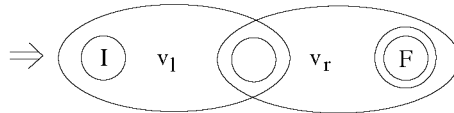
- (i) Construction for the empty word. The automaton consists of just two nodes joined together by an ε -transition.



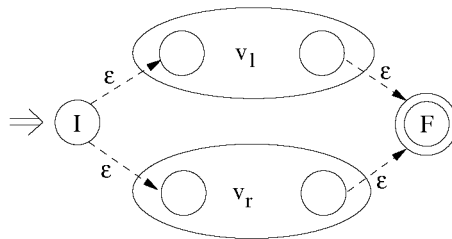
- (ii) For a single character α the construction is similar, except that the transition is labeled with the character rather than with the empty string.



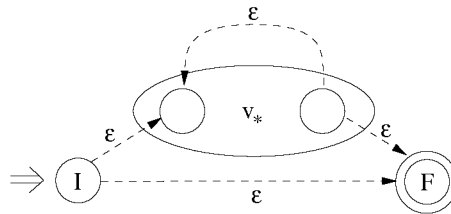
- (iii) Construction for a concatenation node. The two Thompson automata of the two children v_l and v_r are merged together, the final state of the first automaton becoming the initial state of the second.



- (iv) The construction for a union node requires ε -transitions. The idea is to transcript the fact that we enter either automaton $Th(v_l)$ or $Th(v_r)$ of the two children. We then add two new states, an initial one I with two ε -transitions to the two initial states of $Th(v_l)$ and $Th(v_r)$, and a final node F that can be reached from the two final states of $Th(v_l)$ and $Th(v_r)$. A path from I to F has to go through one of the two automata, so the language recognized is $RE_{v_l} \mid RE_{v_r}$.



- (v) The construction for a star node uses the same idea. First, the language RE_{v_*} , where v_* is the only child node of v , now can be repeated as many times as desired. Hence we create a backward ε -transition from the final state of the automaton $Th(v_*)$ to the initial. But the star also means that the automaton $Th(v_*)$ can be ignored, and hence we create two new nodes, an initial I and a final F , joined together by an ε -transition. With two other ε -transitions we join I to the initial state of $Th(v_*)$, and the final state of $Th(v_*)$ to F . The resulting automaton recognizes the language $(RE_{v_*})^*$.



The whole Thompson algorithm consists in performing a bottom-up traversal of the tree representation and keeping the automaton built for the root as the Thompson automaton of the whole expression. The recursive pseudo-code of the algorithm is given in Figure 5.4.

Thompson_recur(v)

1. **If** $v = [\mid] (v_l, v_r)$ **OR** $v = [\cdot] (v_l, v_r)$ **Then**
2. $Th(v_l) \leftarrow \text{Thompson_recur}(v_l)$
3. $Th(v_r) \leftarrow \text{Thompson_recur}(v_r)$
4. **Else If** $v = [*] (v_*)$ **Then** $Th(v_*) \leftarrow \text{Thompson_recur}(v_*)$
5. **End of if**
 /* end of the recursive part, we build the automaton for the current node */
6. **If** $v = (\varepsilon)$ **Then Return** construction (i)
7. **If** $v = (\alpha)$, $\alpha \in \Sigma$ **Then Return** construction (ii)
8. **If** $v = [\cdot] (v_l, v_r)$ **Then Return** construction (iii) on $Th(v_l)$ and $Th(v_r)$
9. **If** $v = [\mid] (v_l, v_r)$ **Then Return** construction (iv) on $Th(v_l)$ and $Th(v_r)$
10. **If** $v = [*] (v_*)$ **Then Return** construction (v) on $Th(v_*)$

Thompson(RE)

11. $v_{RE} \leftarrow \text{Parse}(RE, 1)$ /* parse the regular expression (Section 5.8) */
 12. $Th(v_{RE}) \leftarrow \text{Thompson_recur}(v_{RE})$ /* build the automaton on the tree */
-

Fig. 5.4. The Thompson algorithm. The automaton is built recursively on the tree representation of the expression.

Properties of the Thompson automaton The construction for each node of the tree representation adds at most two states and four transitions to the current automaton. Hence, at the end of the construction, the total number of states and transitions is bounded by $2m$ and $4m$, respectively. We can calculate tighter bounds, but the important point is that the number of states and transitions is linear in m . Moreover, each NFA node has at most two incoming and two outgoing edges, and the whole NFA has one initial and one final state.

Another interesting property is that all the arrows that are not labeled by ε go from states numbered i to states numbered $i + 1$. This is always true provided we process the characters of the regular expression from left to right, as in the parser presented at the end of this chapter.

Complexity The time complexity of the whole algorithm is also linear, since we can create each construction in constant time for each node of the tree representation.

Example of a Thompson automaton construction We build the automaton of $(AT|GA)((AG|AAA)^*)$ from its tree representation (Figure 5.3). The construction is shown in Figure 5.5, except for the basic step of concatenating characters.

5.2.2 Glushkov automaton

The construction of Glushkov [Glu61] has been popularized by Berry and Sethi in [BS86].

We mark the positions of the characters of Σ in RE , counting only characters. For instance, $(AT|GA)((AG|AAA)^*)$ is marked $(A_1T_2|G_3A_4)-((A_5G_6|A_7A_8A_9)^*)$. A *marked expression* from a regular expression RE is denoted \overline{RE} and its language, where each character includes its index, is denoted $L(\overline{RE})$. In our example, $L((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)) = \{A_1T_2, G_3A_4, A_1T_2A_5G_6, G_3A_4A_5G_6, A_1T_2A_7A_8A_9, G_3A_4A_7A_8A_9, A_1T_2A_5G_6A_5G_6, \dots\}$. Let $Pos(\overline{RE}) = \{1 \dots m\}$ be the set of positions in \overline{RE} and $\overline{\Sigma}$ the marked character alphabet.

The Glushkov automaton is built first on the marked expression \overline{RE} and it recognizes $L(\overline{RE})$. We then derive from it the Glushkov automaton that recognizes $L(RE)$ by erasing the position indices of all the characters (see below).

The set of positions is taken as a reference, becoming the set of states of the resulting automaton in addition to an initial state 0. So we build $m + 1$ states labeled from 0 to m . Each state j represents the fact that we have read in the text a string that ends at NFA position j . Now if we read a new character α , we need to know which positions we can reach from j by α . Glushkov computes from a position (state) j all the other accessible positions.

We need four new definitions to explain in depth the algorithm. We denote below by α_y the indexed character of \overline{RE} that is at position y .

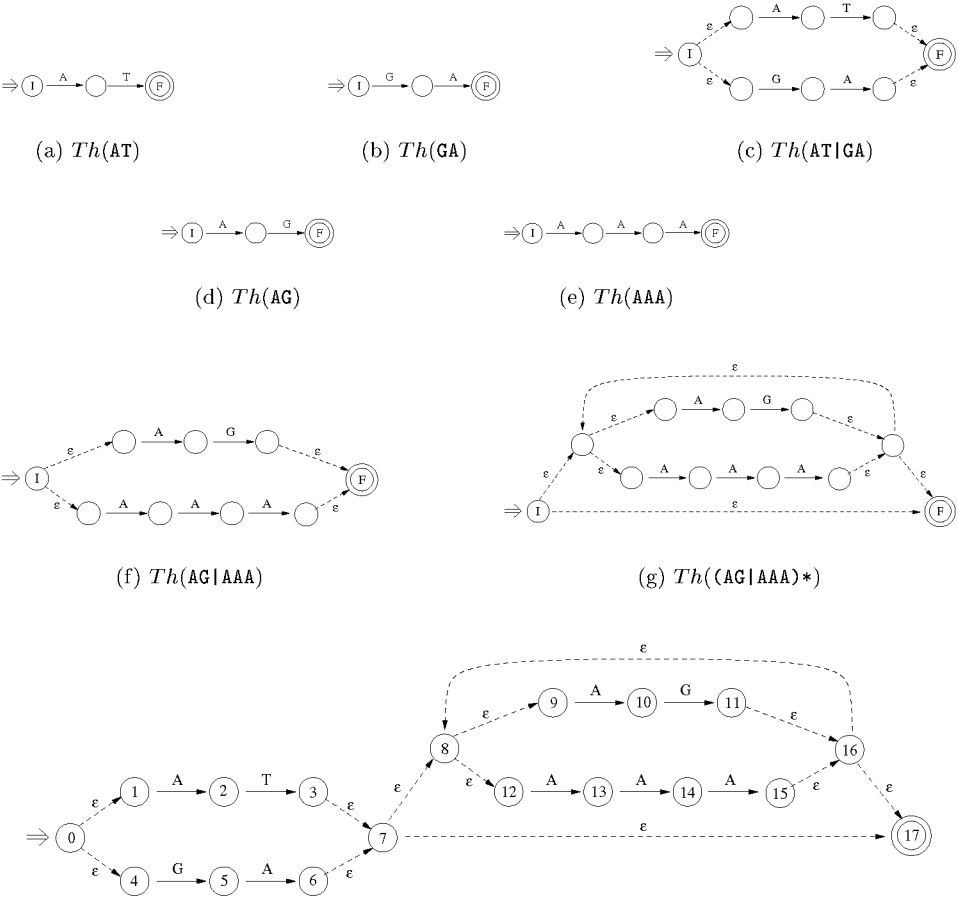


Fig. 5.5. Thompson automaton construction for the regular expression $(AA|AT)((AG|AAA)^*)$.

Definition $First(\overline{RE}) = \{x \in Pos(\overline{RE}), \exists u \in \overline{\Sigma}^*, \alpha_x u \in L(\overline{RE})\}$.

The set $First(\overline{RE})$ represents the set of initial positions of $L(\overline{RE})$, that is, the set of positions at which the reading can start. In our example, $First((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)) = \{1, 3\}$.

Definition $Last(\overline{RE}) = \{x \in Pos(\overline{RE}), \exists u \in \overline{\Sigma}^*, u\alpha_x \in L(\overline{RE})\}$.

The set $Last(\overline{RE})$ represents the set of final positions of $L(\overline{RE})$, that is, the set of positions at which a string read can be recognized. In our example, $Last((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)^*)) = \{2, 4, 6, 9\}$.

Definition $Follow(\overline{RE}, x) = \{y \in Pos(\overline{RE}), \exists u, v \in \overline{\Sigma}^*, u\alpha_x\alpha_yv \in L(\overline{RE})\}$.

The set $Follow(\overline{RE}, x)$ represents all the positions in $Pos(\overline{RE})$ accessible from x . For instance, in our example, if we consider position 6, the set of accessible positions is $Follow((A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*), 6) = \{7, 5\}$.

We need an extra function $Empty_{RE}$ that indicates whether the empty word ε is in $L(RE)$.

Definition We define recursively the function $Empty_{RE}$, whose value is $\{\varepsilon\}$ if ε belongs to $L(RE)$ and \emptyset otherwise.

$$\begin{aligned} Empty_{\varepsilon} &= \{\varepsilon\} \\ Empty_{\alpha \in \Sigma} &= \emptyset \\ Empty_{RE_1|RE_2} &= Empty_{RE_1} \cup Empty_{RE_2} \\ Empty_{RE_1 \cdot RE_2} &= Empty_{RE_1} \cap Empty_{RE_2} \\ Empty_{RE^*} &= \{\varepsilon\} \end{aligned}$$

The deterministic Glushkov automaton \overline{GL} that recognizes the language $L(\overline{RE})$ is built in the following way.

$$\overline{GL} = (S, \Sigma, I, F, \bar{\delta})$$

where:

- (i) S is the set of states, $S = \{0, 1, \dots, m\}$, that is, the set of positions $Pos(\overline{RE})$ and the initial state is $I = 0$.
- (ii) F is the set of final states, $F = Last(\overline{RE}) \cup (Empty_{RE} \cdot \{0\})$. Informally, a state (position) i is final if it is in $Last(\overline{RE})$. The initial state 0 is also final if the empty word ε belongs to $L(\overline{RE})$, in which case $Empty_{RE} = \{\varepsilon\}$ and hence $Empty_{RE} \cdot \{0\} = \{0\}$. If not, $Empty_{RE} \cdot \{0\} = \emptyset$.
- (iii) $\bar{\delta}$ is the transition function of the automaton, defined by

$$\forall x \in Pos(\overline{RE}), \forall y \in Follow(\overline{RE}, x), \bar{\delta}(x, \alpha_y) = y \quad (5.1)$$

Informally, there is a transition from state x to y by α_y if y follows x . The transitions from the initial state are defined by

$$\forall y \in First(\overline{RE}), \bar{\delta}(0, \alpha_y) = y \quad (5.2)$$

The Glushkov automaton of our marked regular expression $(A_1T_2|G_3A_4)((A_5G_6|A_7A_8A_9)*)$ is given in Figure 5.6.

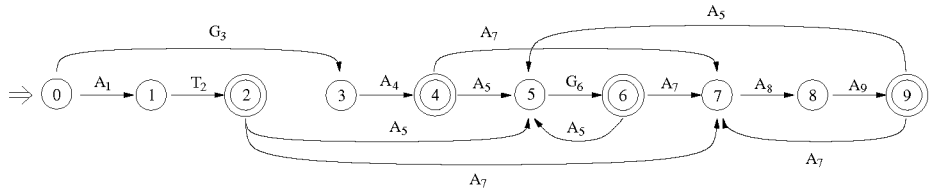


Fig. 5.6. Marked Glushkov automaton built on the marked regular expression $(A_1 T_2 | G_3 A_4) ((A_5 G_6 | A_7 A_8 A_9)^*)$. The state 0 is initial. Double-circled states are final.

To obtain the Glushkov automaton of the original RE , we simply erase the position indices in the marked automaton. At this step, the automaton usually becomes nondeterministic. The new automaton recognizes the language $L(RE)$. The Glushkov automaton of our example $(AT|GA)((AG|AAA)^*)$ is shown in Figure 5.7.

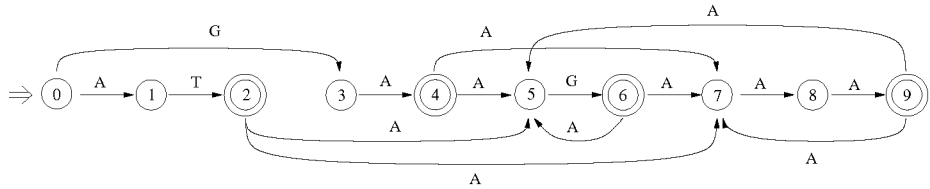


Fig. 5.7. Glushkov automaton built on the regular expression $(AT|GA)((AG|AAA)^*)$. The state 0 is initial. Double-circled states are final. The automaton is derived from the marked automaton by simply erasing the position indices.

The algorithm of Glushkov is based on the tree representation T_{RE} of the regular expression (see Figure 5.3). Each node v of this tree represents a subexpression RE_v of RE . We associate the following variables to v :

- $First(v)$: list of positions that represent the set $First(\overline{RE_v})$.
- $Last(v)$: list of positions that represent the set $Last(\overline{RE_v})$.
- $Empty_v$: set to $\{\varepsilon\}$ if $L(RE_v)$ contains the empty string ε , and to \emptyset otherwise.

These variables are computed for each node in postfix order, that is, they are first computed for every child of v and only afterward for v . We denote the two children of v as v_l and v_r if v is “|” or “.”, and we denote its unique child as v_* if v represents “*”.

The set $Follow(x)$ is a global variable. For each node v we update $Follow(x)$ according to the positions in the subexpression $\overline{RE_v}$.

The recursive algorithm **Glushkov_variables**(v_{RE} , $lpos$) is given in Figure 5.8. It computes the values of $First(v)$, $Last(v)$, $Follow(x)$, and $Empty_v$.

```

Glushkov_variables( $v_{RE}$ ,  $lpos$ )
    /* postfix computation, we compute recursively the children first */
1.  If  $v = [ \mid ] (v_l, v_r)$  OR  $v = [ \cdot ] (v_l, v_r)$  Then
2.       $lpos \leftarrow \mathbf{Glushkov\_variables}(v_l, lpos)$ 
3.       $lpos \leftarrow \mathbf{Glushkov\_variables}(v_r, lpos)$ 
4.  Else If  $v = [ * ] (v_*)$  Then  $lpos \leftarrow \mathbf{Glushkov\_variables}(v_*, lpos)$ 
5.  End of if
    /* end of the recursive part, we compute the values for the current node */
6.  If  $v = (\varepsilon)$  Then
7.       $First(v) \leftarrow \emptyset$ ,  $Last(v) \leftarrow \emptyset$ ,  $Empty_v \leftarrow \{\varepsilon\}$ 
8.  Else If  $v = (\alpha)$ ,  $\alpha \in \Sigma$  Then
9.       $lpos \leftarrow lpos + 1$ 
10.      $First(v) \leftarrow \{lpos\}$ ,  $Last(v) \leftarrow \{lpos\}$ ,  $Empty_v \leftarrow \emptyset$ ,  $Follow(lpos) \leftarrow \emptyset$ 
11.  Else If  $v = [ \mid ] (v_l, v_r)$  Then
12.      $First(v) \leftarrow First(v_l) \cup First(v_r)$ 
13.      $Last(v) \leftarrow Last(v_l) \cup Last(v_r)$ 
14.      $Empty_v \leftarrow Empty_{v_l} \cup Empty_{v_r}$ 
15.  Else If  $v = [ \cdot ] (v_l, v_r)$  Then
16.      $First(v) \leftarrow First(v_l) \cup (Empty_{v_l} \cdot First(v_r))$ ,
17.      $Last(v) \leftarrow (Empty_{v_r} \cdot Last(v_l)) \cup Last(v_r)$ ,
18.      $Empty_v \leftarrow Empty_{v_l} \cap Empty_{v_r}$ 
19.     For  $x \in Last(v_l)$  Do  $Follow(x) \leftarrow Follow(x) \cup First(v_r)$ 
20.  Else If  $v = [ * ] (v_*)$  Then
21.      $First(v) \leftarrow First(v_*)$ ,  $Last(v) \leftarrow Last(v_*)$ ,  $Empty_v \leftarrow \{\varepsilon\}$ 
22.     For  $x \in Last(v_*)$  Do  $Follow(x) \leftarrow Follow(x) \cup First(v_*)$ 
23.  End of if
24.  Return  $lpos$ 

```

Fig. 5.8. Recursive part of the Glushkov algorithm. This function computes the values of $First(v)$, $Last(v)$, $Follow(x)$, and $Empty_v$ for each node v of the tree representation of the regular expression \overline{RE} .

for each node v of the tree representation of the regular expression RE . We visit the nodes in postfix order. The values of the node v_{RE} are computed from the values obtained for its children. The position of each character is computed on the fly (line 9).

The whole Glushkov algorithm consists in transforming RE into a tree v_{RE} , calculating the variables on it with **Glushkov_variables** ($v_{RE}, 0$) and then building the Glushkov automaton from the variables of the root v_{RE} of the tree, following its definition. Pseudo-code for the whole algorithm is given in Figure 5.9.

Properties of the Glushkov automaton Two properties of this automaton are of interest to us. The first one is that the NFA is ε -free. The second

```

Glushkov(RE)
    /* parse the regular expression (Section 5.8) */
1.   $v_{RE} \leftarrow \mathbf{Parse}(RE, 1)$ 
    /* build the variables on the tree */
2.   $m \leftarrow \mathbf{Glushkov\_variables}(v_{RE}, 0)$ 
    /* building the automaton */
3.   $\Delta = \emptyset$ 
4.  For  $i \in 0 \dots m$  Do create state  $i$ 
5.  For  $x \in \mathit{First}(v_{RE})$  Do  $\Delta \leftarrow \Delta \cup \{(0, \alpha_x, x)\}$ 
6.  For  $i \in 0 \dots m$  Do
7.      For  $x \in \mathit{Follow}(i)$  Do  $\Delta \leftarrow \Delta \cup \{(i, \alpha_x, x)\}$ 
8.  End of for
9.  For  $x \in \mathit{Last}(v_{RE}) \cup (\mathit{Empty}_{v_{RE}} \cdot \{0\})$  Do mark  $x$  as terminal

```

Fig. 5.9. The whole Glushkov algorithm. The automaton is nondeterministic in the general case and its transition function is denoted Δ . The initial state is 0.

one is that all the arrows leading to a given state y are labeled by the same character, namely, α_y . This is easily seen in formulas (5.1) and (5.2).

Complexity The worst-case complexity of the whole algorithm is dominated by the function **Glushkov_variables**. In this function, all the unions of sets, except for the star, are disjoint and can be implemented in $O(1)$ time. The **For** loop of line 19 is worst-case $O(m)$. The poor worst-case complexity is due to line 22, that is, the computation of the star. Since $\mathit{Follow}(x)$ and $\mathit{First}(v_*)$ could intersect, the union is worst-case time $O(m)$. As this is inside a **For** loop that can perform $O(m)$ iterations, the whole loop is worst-case time $O(m^2)$. The total complexity of the algorithm is thus worst-case $O(m^3)$, because $O(m)$ stars may exist.

Two variations of this algorithm have been proposed to reduce the worst-case complexity to $O(m^2)$ [BK93, CP92]. Both reduce the complexity of the **For** loop of the star but use different properties. The first one [BK93] uses the fact that

$$\mathit{Follow}(\overline{RE*}, x) = [\mathit{Follow}(\overline{RE*}, x) \setminus \mathit{First}(\overline{RE*})] \cup \mathit{First}(\overline{RE*})$$

while the second [CP92] uses the fact that

$$\mathit{Follow}(\overline{RE*}, x) = \mathit{Follow}(\overline{RE*}, x) \cup [\mathit{First}(\overline{RE*}) \setminus \mathit{Follow}(\overline{RE*}, x)]$$

For our purposes, the $O(m^3)$ time algorithm is good enough, since usually the regular expression is small in comparison to the text size. Moreover, by using bit-parallelism to operate the sets of states, one can obtain $O(m^2 \lceil m/w \rceil)$ time, which is in practice $O(m^2)$ for small regular expressions.

5.3 Classical approaches to regular expression searching

We cover in this section the classical ways to search for a regular expression in a text. We first consider the two extremes: pure NFA and pure DFA simulation. We then introduce a third, intermediate approach, which permits trading space for time.

5.3.1 Thompson's NFA simulation

Together with its NFA definition, Thompson proposed in [Tho68] an $O(mn)$ search algorithm based on the direct simulation of his NFA. The resulting algorithm, which we call **NFAThompson**, is not competitive nowadays, but it is the basis of more competitive algorithms seen later in this chapter.

Thompson stores explicitly the set of currently active states. For each new text character read and for each currently active state, he looks at the new states that the current state activates by this character and adds each of them to a new set of active states. From those new active states he follows all the ε -transitions until all the reachable states are obtained.

Since each state has $O(1)$ outgoing transitions under Thompson's construction and there can be $O(m)$ active states, producing the new set of active states takes $O(m)$ time under a suitable representation of the set of states, for example, a bit vector. The propagation by ε -transitions also takes $O(m)$ time if care is taken to not propagate from a state that was already active. On the other hand, the extra space required is just $O(m)$.

Note that it is possible to use bit-parallelism to store the bit vectors. A smarter use of bit-parallelism is considered in Section 5.4.

5.3.2 Using a deterministic automaton

One of the early achievements in string matching was the $O(n)$ time algorithm to search for a regular expression in a text. As explained, the technique consists of converting the regular expression into a DFA and then searching the text using the DFA. The simplest solution is to build first an NFA with a technique like those shown in the previous sections (e.g., Thompson or Glushkov) and then convert the NFA into a DFA.

This algorithm, which we call **DFAClassical**, can be found in any classical book of compilers, such as [ASU86]. The main idea is as follows. When we traverse the text using a nondeterministic automaton, a number of transitions can be followed and a set of states become active. However, a DFA has exactly one active state at a time. So the corresponding deterministic

automaton is defined over the *set* of states of the nondeterministic automaton. The key idea is that the unique current state of the DFA is the set of current states of the NFA.

To formalize the concepts, we first need a definition.

Definition *The ε -closure of a state s in an NFA, $E(s)$, is the set of states of the NFA that can be reached from s by ε -transitions.*

Note that in ε -free automata like Glushkov's, $E(s) = \{s\}$ for all states s , but this is not true in Thompson's construction.

We can give now a formal definition of the conversion of the NFA into a DFA. Let the NFA be $(Q, \Sigma, I, F, \Delta)$ according to Section 1.3.3. Then the DFA is defined as

$$(\wp(Q), \Sigma, E(I), F', \delta)$$

where

$$F' = \{f \in \wp(Q), f \cap F \neq \emptyset\}$$

and

$$\delta(S, \sigma) = \bigcup_{s', \exists s \in S, (s, \sigma, s') \in \Delta} E(s')$$

that is, for every possible active state s of S we follow all the possible transitions to states s' by the character σ and then follow all the possible ε -transitions from s' .

Since the DFA is built on the set of states of the NFA, its worst-case size is $O(2^m)$ states, which is exponential. This makes the approach suitable for small regular expressions only. In practice, however, most of those states are not reachable from the initial state and therefore do not need to be built.

We now give an algorithm that obtains the DFA from the NFA by building only the reachable states. The algorithm uses sets of NFA states as identifiers for the DFA states. A simple way to represent these sets is to use a boolean array. Note that a bit-parallel representation is also possible, and it permits not only more compact storage but also faster handling of the set union and other required set operations. We give specific bit-parallel algorithms in Section 5.4. For now, we use just an abstract representation of the sets of states.

Figure 5.10 gives pseudo-code to compute the ε -closure $E(s)$ for every state s of the NFA. The result is a set of states for each state s . The algorithm starts with $E(s) = \{s\}$ and then repeatedly traverses the whole automaton looking for ε -transitions. For each of these, it adds the ε -closure

of the target state to that of the source state. The process is repeated until no new information is gathered.

```

EpsClosure( $N = (Q, \Sigma, I, F, \Delta)$ )
1.  For  $s \in Q$  Do  $E(s) \leftarrow \{s\}$ 
2.   $changed \leftarrow \text{TRUE}$ 
3.  While  $changed = \text{TRUE}$  Do
4.     $changed \leftarrow \text{FALSE}$ 
5.    For  $(s, \varepsilon, s') \in \Delta$  Do
6.      If  $E(s') \not\subseteq E(s)$  Then
7.         $E(s) \leftarrow E(s) \cup E(s')$ 
8.         $changed \leftarrow \text{TRUE}$ 
9.      End of if
10.   End of for
11. End of while

```

Fig. 5.10. Computation of the ε -closure $E(s)$.

The cost of this algorithm is $O(|\Delta|m^2)$, since each complete traversal costs $O(|\Delta|m)$ and it adds 1 to the distance up to which the chains of ε -transitions are considered. Since the maximum distance in the NFA is $O(m)$, it follows that $O(m)$ traversals suffice. Under the Thompson construction we know that $|\Delta| \leq 4m$, so the algorithm is $O(m^3)$ time. Under Glushkov we simply do not need to run the algorithm, as we know that $E(s) = \{s\}$ for every $s \in Q$.

Figure 5.11 shows pseudo-code for the algorithm that builds the DFA. The algorithm builds the initial state I_d and then invokes a recursive procedure **BuildState**, which finds all the target states from a given source state and reinvokes itself on all the target states that do not exist yet. The set of final states, F_d , is built together with the set of all states, Q_d .

It is clear that this algorithm produces only the states that are reachable from the initial state, that is, the states that could be reached when reading the text. Its worst-case time complexity is $O(|Q_d||\Sigma||\Delta|\max_s |E(s)|)$, which is $O(|Q_d|m^2)$ on Thompson's NFA since $|\Delta| = O(m)$ as well as on Glushkov's since $|E(s)| = 1$ always.

Example of DFA construction Let us consider our running example $(\text{AT}|\text{GA})(\text{AG}|\text{AAA})^*$. Its Thompson NFA is given in Figure 5.5. Table 5.1 gives the corresponding $E(s)$ function built by **EpsClosure**.

For the Glushkov NFA of Figure 5.7, we have that $E(s) = \{s\}$. Figure 5.12 shows the resulting DFAs from both Thompson's and Glushkov's NFAs. Note that, despite the different labeling, both DFAs are the same. Moreover,

```
BuildState(S)
1.  If  $S \cap F \neq \emptyset$  Then  $F_d \leftarrow F_d \cup \{S\}$ 
2.  For  $\sigma \in \Sigma$  Do
3.     $T \leftarrow \emptyset$ 
4.    For  $s \in S$  Do
5.      For  $(s, \sigma, s') \in \Delta$  Do  $T \leftarrow T \cup E(s')$ 
6.    End of for
7.     $\delta(S, \sigma) \leftarrow T$ 
8.    If  $T \notin Q_d$  Then
9.       $Q_d \leftarrow Q_d \cup \{T\}$ 
10.     BuildState(T)
11.    End of if
12.  End of for

BuildDFA( $N = (Q, \Sigma, I, F, \Delta)$ )
13.  EpsClosure(N)
14.   $I_d \leftarrow E(I)$  /* initial DFA state */
15.   $F_d \leftarrow \emptyset$  /* final DFA states */
16.   $Q_d \leftarrow \{I_d\}$  /* all the DFA states */
17.  BuildState( $I_d$ )
18.  Return  $(Q_d, \Sigma, I_d, F_d, \delta)$ 
```

Fig. 5.11. Classical computation of the DFA from the NFA.

<i>E</i> (0)	{0, 1, 4}	<i>E</i> (9)	{9}
<i>E</i> (1)	{1}	<i>E</i> (10)	{10}
<i>E</i> (2)	{2}	<i>E</i> (11)	{8, 9, 11, 12, 16, 17}
<i>E</i> (3)	{3, 7, 8, 9, 12, 17}	<i>E</i> (12)	{12}
<i>E</i> (4)	{4}	<i>E</i> (13)	{13}
<i>E</i> (5)	{5}	<i>E</i> (14)	{14}
<i>E</i> (6)	{6, 7, 8, 9, 12, 17}	<i>E</i> (15)	{8, 9, 12, 15, 16, 17}
<i>E</i> (7)	{7, 8, 9, 12, 17}	<i>E</i> (16)	{8, 9, 12, 16, 17}
<i>E</i> (8)	{8, 9, 12}	<i>E</i> (17)	{17}

Table 5.1. The ϵ -closure *E*(*s*) for the final NFA of Figure 5.5.

they are minimal, that is, no DFA with fewer states recognizes the same language.

This is not guaranteed in general. Different DFAs may exist to recognize the same language. Moreover, our construction does not guarantee that the result has the minimum size. To ensure this we have to *minimize* the DFA after we build it. Minimization of DFAs is a standard technique that can be found in a classical book such as [ASU86]. We content ourselves with the simple construction, which in most cases produces a DFA of reasonable size.

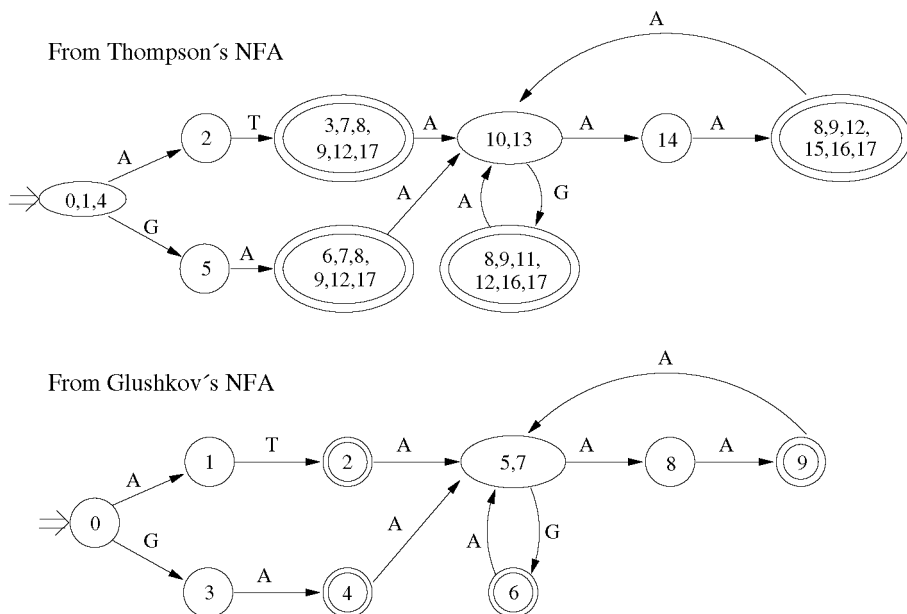


Fig. 5.12. The DFAs resulting from Thompson's and Glushkov's NFAs.

Searching with the DFA The point of building the DFA is to guarantee a linear search time of $O(n)$. This is achievable because we need to cross exactly one transition per text character read. However, we need to modify the automaton in order to use it for text searching. The modification consists of adding a self-loop to the initial state of the NFA, which can be crossed by any character, that is, doing

$$\Delta \leftarrow \Delta \cup \bigcup_{\sigma \in \Sigma} (I, \sigma, I)$$

before converting it into a DFA. If the original automaton recognizes the language $L(RE)$, then after this modification the automaton recognizes $\Sigma^*L(RE)$. Figure 5.13 shows the resulting DFA after adding the self-loop to the Glushkov NFA of Figure 5.7.

The complete search algorithm is depicted in Figure 5.14. The total complexity is $O(m^2 2^m + n)$ in the worst case. The extra space needed to represent the DFA is $O(m 2^m)$ bits.

5.3.3 A hybrid approach

In [Mye92] an approach is proposed which is intermediate between a non-deterministic and a deterministic simulation. The idea is based on Thompson's

is in fact a DFA built on the module with the sets of states represented as bit vectors.

For the lowest level modules it is clear that this DFA can be built. The problem with the higher level modules is that some of their leaves are other submodules. When the bit corresponding to the edge entering the submodule is activated we have to set the initial state of the submodule. And when the final state of the submodule is activated we have to activate the edge leaving the submodule in the higher level module.

Since the construction of modules takes whole subexpressions and Thompson's construction guarantees that there exist just one initial and one final state, the transitions among each module and its parent can be carried out in constant time.

Therefore, to simulate one step of the computation on a higher level module, it is necessary to use the precomputed table to determine which submodules have been reached, and activate their initial state if they have been. Then, we recursively simulate the step on each submodule, and for those that reached their final state we activate the corresponding bit in the higher level module. A final access to the precomputed table yields the final result.

The main problem remaining is the order in which the submodules have to be processed to account for the dependencies between them. Except for the “*” operator, which introduces a *back edge*, the NFA can be processed in topological order (i.e., source nodes before target nodes), and a single pass over the NFA is enough. One of the central points of [Mye92] is to show that two passes in topological order, permitting the source of a back edge to influence its target, are enough to account for all the dependencies. Hence, we need only a constant number of passes over the NFA, working $O(1)$ per module.

Since time is proportional to the number of modules, $O(m/k)$ time suffices to process each text character. Each determinized module needs $O(2^k)$ space to perform all its internal transitions in constant time. Hence we need $O(m2^k/k)$ space and $O(mn/k)$ time. Given $O(s)$ space, the algorithm obtains $O(mn/\log s)$ search time.

5.4 Bit-parallel algorithms

As explained in the previous section, a possible way to store the states of the DFA (i.e., the sets of states of the NFA) is to use a bit mask of $O(m)$ bits where the i -th bit is 1 whenever the i -th NFA state belongs to the DFA state. We present in this section two bit-parallel implementations that are

hybrids between an NFA and a DFA simulation. As we will see later, they have advantages and disadvantages compared to the classical approaches.

Assume that the NFA $(Q = \{s_0 \dots s_{|Q|-1}\}, \Sigma, I = s_0, F, \Delta)$ is represented as follows: $Q_n = \{0 \dots |Q| - 1\}$, $I_n = 0^{|Q|-1}1$, $F_n = \bigvee_{s_j \in F} 0^{|Q|-1-j}10^j$ (i.e., the bitwise OR of the final states positions), and the set of transitions Δ is represented by means of two tables B_n and E_n , where

$$B_n[i, \sigma] = \bigvee_{(s_i, \sigma, s_j) \in \Delta} 0^{|Q|-1-j}10^j$$

represents the states reachable from state i by character σ without considering ε -transitions, and

$$E_n[i] = \bigvee_{s_j \in E(s_i)} 0^{|Q|-1-j}10^j$$

represents $E(i)$, the ε -closure of state s_i (Section 5.3.2).

It is not complicated to produce this representation when applying Thompson's or Glushkov's constructions. Indeed, it is convenient, as we are simply using bit-parallelism to represent sets of states as bit masks of length $|Q|$. Of course E_n is not relevant under Glushkov's construction, since its NFA is ε -free.

5.4.1 Bit-parallel Thompson

A competitive algorithm [WM92b], which we call **BPThompson**, is derived from Thompson's NFA simulation (Section 5.3.1) by a clever use of bit-parallelism. A very important property (Section 5.2.1) is that, except for the ε -transitions, all the arrows go from states numbered i to states numbered $i + 1$.

If we pack the set of states in the bits of a computer word, so that the i -th state is mapped to the i -th bit, then all except the ε -transitions can be simulated using a table B similar to that of the **Shift-And** algorithm (Section 2.2.2). The mechanism to simulate ε -transitions uses a precomputed table E_d . E_d is built such that, for each possible bit mask of active states, it yields the new set of active states that can be reached from the original ones by ε -transitions. This includes the original states and also the initial state 0 and its ε -closure, so as to simulate, without any extra work, the self-loop at the initial state. Formally,

$$E_d[D] = \bigvee_{i, i=0 \text{ OR } D \& 0^{L-i-1}10^i \neq 0^L} E_n[i] \quad (5.3)$$

where $L = |Q| \leq 2m$ is the number of states in Thompson's NFA.

The mechanism is not completely an NFA simulation, since it precomputes a DFA on the ε -transitions. The simulation of all the other transitions can be seen as the true bit-parallel simulation of an NFA.

Figure 5.15 shows the code to build the tables B and E_d . The idea for B is to ignore the originating states of B_n , that is, we store in $B[\sigma]$ all the states that can be reached by the character σ , from any state:

$$B[\sigma] = \bigcup_{i \in 0 \dots m} B_n[i, \sigma] \quad (5.4)$$

The idea for E_d is to iteratively add a new highest bit to the masks and use the results already computed for smaller masks. The overall process takes time $O(2^L + m|\Sigma|)$.

```

BuildEps( $N = (Q_n, \Sigma, I_n, F_n, B_n, E_n)$ )
1.  For  $\sigma \in \Sigma$  Do
2.     $B[\sigma] \leftarrow 0^L$ 
3.    For  $i \in 0 \dots L - 1$  Do  $B[\sigma] \leftarrow B[\sigma] \mid B_n[i, \sigma]$ 
4.  End of for
      /*  $B$  is already built, now build  $E_d$  */
5.   $E_d[0] \leftarrow E_n[0]$  /* the initial state and its closure */
6.  For  $i \in 0 \dots L - 1$  Do
7.    For  $j \in 0 \dots 2^i - 1$  Do /* recall that  $E_n[i]$  includes  $i$  */
8.       $E_d[2^i + j] \leftarrow E_n[i] \mid E_d[j]$ 
9.    End of for
10. End of for
11. Return ( $B, E_d$ )

```

Fig. 5.15. Bit-parallel construction of E_d and B from Thompson's NFA. We use a numeric notation for the arguments of E_d .

Figure 5.16 shows the search algorithm. Each transition is simulated in two steps: First we use a **Shift-And**-like mechanism for the normal transitions using B , and second we use E_d to simulate all the ε -transitions.

Reducing space A table of size 2^L may be too large depending on the machine and the pattern. However, a *horizontal partitioning* scheme can be used to fit the available memory. We split E_d into two tables, E_d^1 and E_d^2 , each of them defined over half of the bits. This exploits the following property, which comes directly from equation (5.3):

$$E_d[D_1 D_2] = E_d[D_1 0^{|D_2|}] \mid E_d[0^{|D_1|} D_2]$$

```

BPTompson( $N = (Q_n, \Sigma, I_n, F_n, B_n, E_n), T = t_1 t_2 \dots t_n$ )
1.  Preprocessing
2.     $(B, E_d) \leftarrow \mathbf{BuildEps}(N)$ 
3.  Searching
4.     $D \leftarrow E_d[I_n]$  /* the initial state */
5.    For  $pos \in 1 \dots n$  Do
6.      If  $D \ \& \ F_n \neq 0^L$  Then report an occurrence ending at  $pos - 1$ 
7.       $D \leftarrow E_d[(D < 1) \ \& \ B[t_{pos}]]$ 
8.    End of for

```

Fig. 5.16. Thompson's bit-parallel search algorithm.

that is, we can decompose the argument of E_d in two parts. Hence E_d^1 and E_d^2 are defined as follows, over masks of length $\lfloor L/2 \rfloor$ and $\lceil L/2 \rceil$, respectively:

$$E_d^1[D] = E_d[0^{\lceil L/2 \rceil} D], \quad E_d^2[D] = E_d[D 0^{\lfloor L/2 \rfloor}]$$

and hence it holds

$$E_d[d_m \dots d_0] = E_d^1[d_{\lfloor L/2 \rfloor - 1} \dots d_0] \mid E_d^2[d_m \dots d_{\lfloor L/2 \rfloor}]$$

For instance, in Figure 5.5 we would have $E_n[3] = 100001001110001000$ and $E_n[11] = 111001101100000000$, so $E_d^1[000001000] = 100001001110001000$ and $E_d^2[000000100] = 111001101100000000$. Thus, $E_d[000000100000001000] = 111001101110001000$.

The net result is that, instead of having a table of size $O(2^L)$, we have two much smaller tables, of size $O(2^{L/2})$. The cost is that we have to pay two accesses to memory in order to perform each transition.

The scheme can be generalized as follows. Assume that we have $O(s)$ space available for the tables. We split our table E_d into k tables $E_d^1 \dots E_d^k$, each one addressing $\lfloor L/k \rfloor$ or $\lceil L/k \rceil$ bits of the argument mask. The total space required is $O(k2^{L/k})$. If this space is s , then we have that $k \approx L/\log_2 s$. Therefore, the scheme permits a search time of $O(mn/\log s)$ using $O(s)$ space. This trade-off cannot be achieved with the classical DFA algorithm. Note that the complexity has to be multiplied by m/w for long patterns.

Depending on the architecture, even when a large table fits in memory, the cache optimization mechanism can make it advisable to use two smaller tables, which have more locality of reference.

Example of BPTompson We search for the pattern $(\mathbf{AT|GA})(\mathbf{AG|AAA})^*$ in the text **AAAGATAAGATAGAAAA**, marking the final positions of occurrences. The states have been numbered according to Figure 5.5. As it is not

practical to show the whole table E_d of $2^{18} = 262,144$ entries, we show the table E_n . Remember that the E_d rows are obtained by OR-ing the E_n rows corresponding to the bits set in the argument of E_d . We only show the E_n entries where $E(s) \neq \{s\}$; otherwise $E_n[s]$ contains $E(0) \cup \{s\}$.

For each character read we show two steps in the update of D , namely, before and after the ε -closure.

Table E_n	
0	0000000000000000010011
3	10000100111001101011
6	10000100111101001011
7	10000100111001001011
8	00000100110001001011
11	11000110110001001011
15	11100100110001001011
16	11000100110001001011

A	001110010001000100
C	000000000000000000
G	000000100000100000
T	000000000000000100
*	000000000000000000

Table B

$$F_n = 100000000000000000$$

$$D = 0000000000000010011$$

1. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 000000000000000100 \\ D = \quad 0000000000000010111 \end{array}$$

2. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 000000000000000100 \\ D = \quad 0000000000000010111 \end{array}$$

3. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 000000000000000100 \\ D = \quad 0000000000000010111 \end{array}$$

4. Reading G

$$\begin{array}{r} B[G] \quad 000000100000100000 \\ D = \quad 000000000000010000 \\ D = \quad 0000000000000110011 \end{array}$$

5. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 000000000001000100 \\ D = \quad 100001001111010111 \end{array}$$

$D \& F_n \neq 0^L$, so we mark an occurrence.

6. Reading T

$$\begin{array}{r} B[T] \quad 0000000000000001000 \\ D = \quad 0000000000000001000 \\ D = \quad 1000010011100110111 \end{array}$$

$D \& F_n \neq 0^L$, so we mark an occurrence.

7. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 0000100100000000100 \\ D = \quad 0000100100000010111 \end{array}$$

8. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 0001000000000000100 \\ D = \quad 0001000000000010111 \end{array}$$

9. Reading G

$$\begin{array}{r} B[G] \quad 000000100000100000 \\ D = \quad 000000000000010000 \\ D = \quad 0000000000000110011 \end{array}$$

10. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 000000000001000100 \\ D = \quad 100001001111010111 \end{array}$$

$D \& F_n \neq 0^L$, so we mark an occurrence.

11. Reading T

$$\begin{array}{r} B[T] \quad 0000000000000001000 \\ D = \quad 0000000000000001000 \\ D = \quad 1000010011100110111 \end{array}$$

$D \& F_n \neq 0^L$, so we mark an occurrence.

12. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ D = \quad 0000100100000000100 \\ D = \quad 0000100100000010111 \end{array}$$

13. Reading G

$$\begin{array}{r} B[G] \quad 000000100000100000 \\ \hline D = \quad 000000100000100000 \\ D = \quad 110001101100110011 \end{array}$$

 $D \& F_n \neq 0^L$, so we mark an occurrence.

16. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ \hline D = \quad 001100000000000100 \\ D = \quad 111101001100010111 \end{array}$$

 $D \& F_n \neq 0^L$, so we mark an occurrence.

14. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ \hline D = \quad 000010010001000100 \\ D = \quad 100011011111010111 \end{array}$$

 $D \& F_n \neq 0^L$, so we mark an occurrence.

17. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ \hline D = \quad 001010010000000100 \\ D = \quad 111011011100010111 \end{array}$$

 $D \& F_n \neq 0^L$, so we mark an occurrence.

15. Reading A

$$\begin{array}{r} B[A] \quad 001110010001000100 \\ \hline D = \quad 000110010000000100 \\ D = \quad 000110010000010111 \end{array}$$

5.4.2 Bit-parallel Glushkov

Another bit-parallel algorithm [NR99a, Nav01b, NR01a] uses Glushkov's NFA, which has exactly $m + 1$ states. We call it **BP**Glushkov.

The reason to choose Glushkov over Thompson is that we need to build and store a table whose size is $2^{|Q|}$, and Thompson's automaton has more states than Glushkov's. The price is that now the transitions of the automaton cannot be decomposed into forward ones plus ε -transitions. In Glushkov's construction there are no ε -transitions, but the transitions by characters do not follow a simple forward pattern.

However, there is another property enforced by Glushkov's construction that can be successfully exploited (Section 5.2.2): All the arrows arriving at a given state are labeled by the same character. So we can compute the transitions by using two tables: $B[\sigma]$ (formula (5.4)) tells which states can be reached by character σ , and

$$T_d[D] = |(i, \sigma), D \& 0^{m-i} 1 0^i \neq 0^{m+1}, \sigma \in \Sigma| B_n[i, \sigma]$$

tells which states can be reached from D by any character.

Thus $\delta(D, \sigma) = T_d[D] \& B[\sigma]$. We use this property to build and store only T_d and B instead of a complete transition table. Figure 5.17 shows the necessary preprocessing. The ideas are similar to those used to build E_d and B in Section 5.4.1. This time the cost is $O(2^m + m|\Sigma|)$ by using an intermediate table $A[i] = |\sigma \in \Sigma| B[i, \sigma]$, which is essentially a bit-parallel

representation of the *Follow* set (Section 5.2.2). Figure 5.18 shows the search algorithm, which is similar to **BPTompson**.

```

BuildTran ( $N = (Q_n, \Sigma, I_n, F_n, B_n)$ )
1.  For  $i \in 0 \dots m$  Do  $A[i] \leftarrow 0^{m+1}$ 
2.  For  $\sigma \in \Sigma$  Do  $B[\sigma] \leftarrow 0^{m+1}$ 
3.  For  $i \in 0 \dots m, \sigma \in \Sigma$  Do
4.       $A[i] \leftarrow A[i] \mid B_n[i, \sigma]$ 
5.       $B[\sigma] \leftarrow B[\sigma] \mid B_n[i, \sigma]$ 
6.  End of for
    /*  $B$  and  $A$  are built, now build  $T_d$  */
7.   $T_d[0] \leftarrow 0^{m+1}$ 
8.  For  $i \in 0 \dots m$  Do
9.      For  $j \in 0 \dots 2^i - 1$  Do
10.          $T_d[2^i + j] \leftarrow A[i] \mid T_d[j]$ 
11.     End of for
12. End of for
13. Return ( $B, T_d$ )

```

Fig. 5.17. Bit-parallel construction of B and T_d from Glushkov's NFA. We use a numeric notation for the argument of T_d .

```

BPGlushkov( $N = (Q_n, \Sigma, I_n, F_n, B_n), T = t_1 t_2 \dots t_n$ )
1.  Preprocessing
2.      For  $\sigma \in \Sigma$  Do  $B_n[0, \sigma] \leftarrow B_n[0, \sigma] \mid 0^m 1$  /* initial self-loop */
3.       $(B, T_d) \leftarrow \text{BuildTran}(N)$ 
4.  Searching
5.       $D \leftarrow 0^m 1$  /* the initial state */
6.      For  $pos \in 1 \dots n$  Do
7.          If  $D \ \& \ F_n \neq 0^{m+1}$  Then report an occurrence ending at  $pos - 1$ 
8.           $D \leftarrow T_d[D] \ \& \ B[t_{pos}]$ 
9.      End of for

```

Fig. 5.18. Glushkov's bit-parallel search algorithm.

Compared to **BPTompson**, **BPGlushkov** has the advantage of needing $O(2^m)$ space instead of up to $O(2^{2m})$. Just as for E_d , it is possible to split T_d horizontally to obtain $O(mn/\log s)$ time with $O(s)$ space. Therefore, **BPGlushkov** should be always preferred over **BPTompson**.

Example of BPGlushkov We search for the pattern $(AT|GA)((AG|AAA)^*)$ in the text **AAAGATAAGATAGAAAA**, marking the final position of occurrences. We use Glushkov's simulation, where the states have been numbered ac-

cording to Figure 5.7. Since it is not practical to show the whole table T_d of $2^{10} = 1024$ entries, we show only the tables B_n , B , and the rows of T_d that are needed in the search. Remember that the T_d rows are obtained by OR-ing the B_n rows corresponding to the bits set in the argument of T_d over every character. In B_n we only show the entries leading to a nonzero result.

$$B_n = \begin{Bmatrix} \begin{array}{|c|c|c|} \hline 0 & A & 0000000011 \\ \hline 0 & C & 0000000001 \\ \hline 0 & G & 00000001001 \\ \hline 0 & T & 0000000001 \\ \hline 1 & T & 00000000100 \\ \hline 2 & A & 0010100000 \\ \hline 3 & A & 00000010000 \\ \hline 4 & A & 0010100000 \\ \hline 5 & G & 0001000000 \\ \hline 6 & A & 0010100000 \\ \hline 7 & A & 0100000000 \\ \hline 8 & A & 1000000000 \\ \hline 9 & A & 0010100000 \\ \hline \end{array} \\ \begin{array}{|c|c|c|} \hline A & 1110110011 \\ \hline C & 0000000001 \\ \hline G & 0001001001 \\ \hline T & 0000000101 \\ \hline \end{array} \end{Bmatrix}$$

$$F_n = 1001010100 \\ D = 0000000001$$

$$\begin{array}{l} 1. \text{ Reading A} \\ \hline T_d[D] = 0000001011 \\ B[A] = 1110110011 \\ D = 0000000011 \end{array}$$

$$\begin{array}{l} 2. \text{ Reading A} \\ \hline T_d[D] = 0000001111 \\ B[A] = 1110110011 \\ D = 0000000011 \end{array}$$

$$\begin{array}{l} 3. \text{ Reading A} \\ \hline T_d[D] = 0000001111 \\ B[A] = 1110110011 \\ D = 0000000011 \end{array}$$

$$\begin{array}{l} 4. \text{ Reading G} \\ \hline T_d[D] = 0000001111 \\ B[G] = 0001001001 \\ D = 0000001001 \end{array}$$

$$\begin{array}{l} 5. \text{ Reading A} \\ \hline T_d[D] = 0000011011 \\ B[A] = 1110110011 \\ D = 0000010011 \end{array}$$

$D \& F_n \neq 0^{m+1}$, so we mark an occurrence.

$$\begin{array}{l} 6. \text{ Reading T} \\ \hline T_d[D] = 0010101111 \\ B[T] = 0000000101 \\ D = 0000000101 \end{array}$$

$D \& F_n \neq 0^{m+1}$, so we mark an occurrence.

$$\begin{array}{l} 7. \text{ Reading A} \\ \hline T_d[D] = 0010101011 \\ B[A] = 1110110011 \\ D = 0010100011 \end{array}$$

$$\begin{array}{l} 8. \text{ Reading A} \\ \hline T_d[D] = 0101001111 \\ B[A] = 1110110011 \\ D = 0100000011 \end{array}$$

$$\begin{array}{l} 9. \text{ Reading G} \\ \hline T_d[D] = 1000001111 \\ B[G] = 0001001001 \\ D = 0000001001 \end{array}$$

$$\begin{array}{l} 10. \text{ Reading A} \\ \hline T_d[D] = 0000011011 \\ B[A] = 1110110011 \\ D = 0000010011 \end{array}$$

$D \& F_n \neq 0^{m+1}$, so we mark an occurrence.

$$\begin{array}{l} 11. \text{ Reading T} \\ \hline T_d[D] = 0010101111 \\ B[T] = 0000000101 \\ D = 0000000101 \end{array}$$

$D \& F_n \neq 0^{m+1}$, so we mark an occurrence.

$$\begin{array}{l} 12. \text{ Reading A} \\ \hline T_d[D] = 0010101011 \\ B[A] = 1110110011 \\ D = 0010100011 \end{array}$$

Given a regular expression, we compute the length ℓ_{min} of its shortest occurrence. Any method based on skipping text characters must examine at least one out of every ℓ_{min} characters to avoid missing an occurrence. Hence, in general we will use a window of length ℓ_{min} .

Figure 5.20 gives the recursive algorithm to compute ℓ_{min} in $O(m)$ time using the parse tree of the regular expression. A shortest path algorithm from the initial to a final NFA state is also possible.

```

Lmin( $v$ )
1.  If  $v = [ | ] (v_l, v_r)$  Then Return  $\min(\mathbf{Lmin}(v_l), \mathbf{Lmin}(v_r))$ 
2.  If  $v = [ \cdot ] (v_l, v_r)$  Then Return  $\mathbf{Lmin}(v_l) + \mathbf{Lmin}(v_r)$ 
3.  If  $v = [ * ] (v_*)$  Then Return 0
4.  If  $v = ( \alpha )$ ,  $\alpha \in \Sigma$  Then Return 1
5.  If  $v = ( \varepsilon )$  Then Return 0

```

Fig. 5.20. Computation of ℓ_{min} .

5.5.1 Multistring matching approach

This method [Wat96], which we call **MultiStringRE**, consists of generating the prefixes of length ℓ_{min} for all the strings matching the regular expression $Pref(RE)$. In the regular expression $RE = ((GA|AAA)*) (TA|AG)$ we have $\ell_{min}(RE) = 2$, and the set of length-2 prefixes of strings matching the pattern is $Pref(RE) = \{ GA, AA, TA, AG \}$. A more complex example would be $RE = (AT|GA) (AG|AAA) ((AG|AAA)+)$, where $\ell_{min}(RE) = 6$ and the set of prefixes is $Pref(RE) = \{ ATAGAG, ATAGAA, ATAAAA, GAAGAG, GAAGAA, GAAAAA \}$.

Figure 5.21 gives pseudo-code that generates the set of prefixes from a regular expression. A very convenient way of representing $Pref$ is as a trie, because it is easier to generate and to use later for searching. For simplicity we assume that the NFA is ε -free. The time is worst-case $O(|\Delta|^{\ell_{min}})$.

For reasons that will become clear soon, we also store at each trie leaf x the DFA state $Active(x)$ that is reached by reading each trie path. In this case we represent the DFA state as the set of NFA states. It is also possible to write a version of **Compute_Pref** that works on the DFA, and in this case any other representation for DFA states can be used as well.

Once the set of prefixes is computed, the algorithm uses a multipattern search for the set $Pref(RE)$ (Chapter 3). In particular, [Wat96] focuses on **Commentz-Walter**-like algorithms. Since every occurrence of the regular

```

Pref( $s, \Delta, \ell_{min}, Trie$ )
1.  If  $\ell_{min} = 0$  Then /* trie leaf */
2.       $Active(Trie) \leftarrow Active(Trie) \cup \{s\}$ 
3.      Return  $Trie$ 
4.  End of if
5.  For  $(s, \sigma, s') \in \Delta$  Do
6.      If  $\delta(Trie, \sigma) = \theta$  Then
7.          Create new state  $Next = \delta(Trie, \sigma)$ 
8.           $Active(Next) \leftarrow \emptyset$ 
9.      End of if
10.     Pref( $s', \Delta, \ell_{min} - 1, Next$ )
11. End of for

Compute_Pref( $N = (Q, \Sigma, I, F, \Delta), \ell_{min}$ )
12.  $Trie \leftarrow \theta$ 
13. Pref( $I, \Delta, \ell_{min}, Trie$ )
14. Return ( $Trie, Active$ )

```

Fig. 5.21. Computation of *Pref*. It receives an ε -free NFA and ℓ_{min} and returns *Pref* in trie form and *Active* at the leaves.

expression must start with the occurrence of a string in $Pref(RE)$, it is enough to check for the occurrences of RE that start at the initial positions of $Pref(RE)$ in the text. To check for an occurrence starting at a given position we can use any of the methods seen earlier in this chapter, except that we do *not* add the initial self-loop. This forces the occurrence to start at the position specified. Since the length of a string matching a regular expression is in general unbounded, we have to run the automaton until it reaches a final state, it runs out of active states, or we reach the end of the text.

To avoid re-reading the first ℓ_{min} characters of the window at verification time, we initialize the automaton with the states in $Active(x)$ and start reading the characters after the window. In particular, if we use a bit-parallel representation of the DFA, then *Active* can be stored as a bit mask and used directly to initialize the automaton.

The effectiveness of this method depends basically on two values: ℓ_{min} (the search is faster for larger ℓ_{min}) and the size of $Pref(RE)$ (the search is faster for less prefixes). Note that the size of $Pref(RE)$ can be exponential in m , for example, searching for $(a|b)(a|b) \dots (a|b)$. It is possible to artificially reduce ℓ_{min} to avoid an excessively large trie. We see in Section 5.5.3 a method that avoids this problem.

```
MultiStringRE( $N = (Q, \Sigma, I, F, \Delta)$ ,  $\ell_{min}$ )
1.  Preprocessing
    /* Construction of Pref */
2.    (Pref, Active)  $\leftarrow$  Compute_Pref( $N, \ell_{min}$ )
    /* Construction of the DFA (Figure 5.17) without initial self-loop */
3.    Produce bit-parallel version  $N' = (Q_n, \Sigma, I_n, F_n, B_n)$  of  $N$ 
4.    ( $B, T_d$ )  $\leftarrow$  BuildTran( $N'$ )
5.  Searching
    /* Multipattern search of Pref. Check each occurrence with the DFA */
6.    For ( $pos, i$ )  $\in$  output of multipattern search of Pref Do
7.       $D \leftarrow Active(i)$ ,  $j \leftarrow pos + 1$ 
8.      While  $j \leq n$  AND  $D \& F_n = 0^{m+1}$  AND  $D \neq 0^{m+1}$  Do
9.         $D \leftarrow T_d[D] \& B[t_j]$ 
10.     End of while
11.     If  $D \& F_n \neq 0^{m+1}$  Then
12.       Report an occurrence beginning at  $pos + 1 - \ell_{min}$ 
13.     End of if
14.  End of for
```

Fig. 5.22. **MultiStringRE** search algorithm. It receives an NFA and the minimum length of a string accepted by it and reports the initial positions of occurrences. We assume that the verification is done with the bit-parallel Glushkov simulation of Section 5.4.2. Consequently, we assume a bit map representation of *Active*.

Example of MultiStringRE search We search for the pattern $((GA|AAA)*) (TA|AG)$ in the text AAAAGATAGATAAGAAA, the reverse of the example text used earlier in this chapter, and mark the initial positions of occurrences.

We use as our verification engine the bit-parallel Glushkov simulation of Section 5.4.2, where the states have been numbered according to Figure 5.19. As before, we only show the nonzero B_n entries.

The example may look clumsy because our search pattern and text permit little filtering. However, the example shows all the cases that may occur.

$$B_n = \left\{ \begin{array}{|c|c|c|} \hline 0 & A & 01000001000 \\ \hline 0 & G & 00000000010 \\ \hline 0 & T & 00010000000 \\ \hline 1 & A & 00000000100 \\ \hline 2 & A & 01000001000 \\ \hline 2 & G & 00000000010 \\ \hline 2 & T & 00010000000 \\ \hline 3 & A & 00000010000 \\ \hline 4 & A & 00000100000 \\ \hline 5 & A & 01000001000 \\ \hline 5 & G & 00000000010 \\ \hline 5 & T & 00010000000 \\ \hline 6 & A & 00100000000 \\ \hline 8 & G & 10000000000 \\ \hline \end{array} \right.$$

$$B = \left\{ \begin{array}{|c|c|} \hline A & 0110111100 \\ \hline C & 0000000000 \\ \hline G & 1000000010 \\ \hline T & 0001000000 \\ \hline \end{array} \right.$$

$$Pref = \left\{ \begin{array}{|c|c|} \hline \text{prefix} & Active \\ \hline GA & 0000000100 \\ \hline AA & 0000010000 \\ \hline TA & 0010000000 \\ \hline AG & 1000000000 \\ \hline \end{array} \right.$$

$$F_n = 1010000000$$

$$\ell_{min} = 2$$

- 1.
- \boxed{AA}
- AAGATAGAATAGAAA

$$\begin{array}{r} D = 0000010000 \\ \hline \text{Reading A} \ 0000100000 \\ \text{Reading A} \ 0100001000 \\ \text{Reading G} \ 1000000000 \end{array}$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 1.

2. A
- \boxed{AA}
- AGATAGAATAGAAA

$$\begin{array}{r} D = 0000010000 \\ \hline \text{Reading A} \ 0000100000 \\ \text{Reading G} \ 0000000010 \\ \text{Reading A} \ 0000000100 \\ \text{Reading T} \ 0001000000 \\ \text{Reading A} \ 0010000000 \end{array}$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 2.

3. AA
- \boxed{AA}
- GATAGAATAGAAA

$$\begin{array}{r} D = 0000010000 \\ \hline \text{Reading G} \ 0000000000 \end{array}$$

$D = 0^{m+1}$, so we discard position 3.

4. AAA
- \boxed{AG}
- ATAGAATAGAAA

$$D = 1000000000$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 4.

5. AAAA
- \boxed{GA}
- TAGAATAGAAA

$$\begin{array}{r} D = 0000000100 \\ \hline \text{Reading T} \ 0001000000 \\ \text{Reading A} \ 0010000000 \end{array}$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 5.

6. AAAAGA
- \boxed{TA}
- GAATAGAAA
-
- (we skipped position 6).

$$D = 0010000000$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 7.

7. AAAAGAT
- \boxed{AG}
- AATAGAAA

$$D = 1000000000$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 8.

8. AAAAGATA
- \boxed{GA}
- ATAGAAA

$$\begin{array}{r} D = 0000000100 \\ \hline \text{Reading A} \ 0100001000 \\ \text{Reading T} \ 0000000000 \end{array}$$

$D = 0^{m+1}$, so we discard position 9.

9. AAAAGATAG
- \boxed{AA}
- TAGAAA

$$\begin{array}{r} D = 0000010000 \\ \hline \text{Reading T} \ 0000000000 \end{array}$$

$D = 0^{m+1}$, so we discard position 10.

10. AAAAGATAGAA
- \boxed{TA}
- GAAA

$$D = 0010000000$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 12.

11. AAAAGATAGAAAT
- \boxed{AG}
- AAA

$$D = 1000000000$$

$D \&F_n \neq 0^{m+1}$, so we report an occurrence beginning at 13.

12. AAAAGATAGAAATA
- \boxed{GA}
- AA

$$\begin{array}{r} D = 0000000100 \\ \hline \text{Reading A} \ 0100001000 \\ \text{Reading A} \ 0000010000 \end{array}$$

The text finishes without an occurrence, so we discard text position 14.

13. AAAAGATAGAAATAG
- \boxed{AA}
- A

$$\begin{array}{r} D = 0000010000 \\ \hline \text{Reading A} \ 0000100000 \end{array}$$

The text finishes without an occurrence, so we discard text position 15.

14. AAAAGATAGAAATAGA
- \boxed{AA}

$$D = 0000010000$$

The text finishes without an occurrence, so we discard text position 16.

5.5.2 Gnu's heuristic based on necessary factors

A heuristic used in *Gnu Grep* consists of selecting a *necessary set of factors*. We call it **MultiFactRE**. In the simplest case, we may find that a given string must appear in every occurrence of the regular expression. For example, if we look for $(AG|GA)ATA((TT)^*)$, then the string *ATA* is a necessary factor.

The idea in general is to find a set of necessary factors and perform a multipattern search for all of them. There are many ways to choose a suitable set, and *Grep*'s documentation is insufficient to determine its technique. Note that *Pref* is just a particular case of this approach. The advantage of *Pref* is that we know where the match should start, while the general method may need a verification in both directions starting from the factor found.

The selection of the best set of necessary factors has two parts. The first part is an algorithm that detects the correct candidate sets. The second part is a function that evaluates the cost to search using a candidate set and the number of potential matches it produces. A good measure for evaluating a set is its overall probability of occurrence, but finer considerations may include knowledge of the search algorithm used.

Figure 5.23 gives an algorithm that finds sets of necessary factors and selects the best one, assuming that a function *best* to compare sets has been defined. The code works recursively on the parse tree of the regular expression and returns $(all, pref, suff, fact)$, where *all* is the set of all the strings matching the expression, *pref* is the best set of prefixes, *suff* is the best set of suffixes, and *fact* is the best set of factors. Our answer is the fourth element of the tuple returned. If this is θ , then no finite set of necessary factors exists.

The easiest cases are single characters and ε . For a “*” operator, the strings inside can be repeated an unbounded number of times, so we cannot guarantee a finite set for *all*. So we return $\{\varepsilon\}$ for *pref*, *suff* and *fact*, and θ for *all*. For a “|” operator, we need to make the union of the two children for each of the four values. Note that we have to keep any θ present at the children. Finally, the most interesting operator is “.”. To obtain $all(RE_1RE_2)$ we concatenate any string of $all(RE_1)$ to any string of $all(RE_2)$. To obtain the best $pref(RE_1RE_2)$ we choose the best among $pref(RE_1)$ and $pref(RE_2)$, with the understanding that this last set has to be preceded by $all(RE_1)$. The case of *suff* is symmetrical. Finally, for $fact(RE_1RE_2)$ we can choose between $fact(RE_1)$, $fact(RE_2)$, and $suff(RE_1)$ concatenated to $pref(RE_2)$.

```

BestFactor( $v$ )
1.  If  $v = [ \ ] (v_l, v_r)$  OR  $v = \boxed{\cdot} (v_l, v_r)$  Then
2.       $(all_l, pref_l, suff_l, fact_l) \leftarrow \mathbf{BestFactor}(v_l)$ 
3.       $(all_r, pref_r, suff_r, fact_r) \leftarrow \mathbf{BestFactor}(v_r)$ 
4.  End of if
5.  If  $v = [ \ ] (v_l, v_r)$  Then
6.      Return  $(all_l \cup all_r, pref_l \cup pref_r, suff_l \cup suff_r, fact_l \cup fact_r)$ 
7.  Else If  $v = \boxed{\cdot} (v_l, v_r)$  Then
8.      Return  $(all_l \cdot all_r, best(pref_l, all_l \cdot pref_r),$ 
            $best(suff_r, suff_l \cdot all_r), best(fact_l, fact_r, suff_l \cdot pref_r))$ 
9.  Else If  $v = \boxed{*} (v_*)$  Then Return  $(\theta, \theta, \theta, \theta)$ 
10. Else If  $v = (\alpha)$ ,  $\alpha \in \Sigma$  Then Return  $(\{\alpha\}, \{\alpha\}, \{\alpha\}, \{\alpha\})$ 
11. Else If  $v = (\varepsilon)$  Then Return  $(\{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\}, \{\varepsilon\})$ 
12. End of if

```

Fig. 5.23. Computation of the best set of necessary factors. We assume that θ acts as Σ^* , so that $\theta \cup A = A \cup \theta = \theta \cdot A = A \cdot \theta = \theta$ for any A . Also, *best* always considers θ the worst option.

This method gives better results than **MultiStringRE** because it has the potential of choosing the best set. In the example $((GA|AAA)^*)(TA|AG)$, instead of choosing a set of four strings as **MultiStringRE** does, it can choose $\{TA, AG\}$, which is smaller.

5.5.3 An approach based on BNDM

Our final technique able to skip characters [NR99a, Nav01b] is an extension of **BNDM** (Sections 2.4.2, 4.3.2, and 4.2.2) to regular expressions. We call it **RegularBNDM**. It has the benefit of using the same space as a forward search.

The idea is based on the bit-parallel DFA simulation of Glushkov's construction (Section 5.4.2). We modify the DFA by reversing the arrows and making all states initial, so that the resulting automaton recognizes every reverse prefix of RE and is alive as long as we have read a reverse factor of RE . Note that this automaton does not have an initial self-loop. Figure 5.24 shows the result on $((GA|AAA)^*)(TA|AG)$.

We slide a window of length ℓ_{min} along the text. The window is read backwards with the automaton. Each time we recognize a prefix we store in a variable *last* the window position where this happened. When the window is shifted, it is aligned so as to start at position *last*. The backward traversal inside the window may finish because the DFA runs out of active states, in

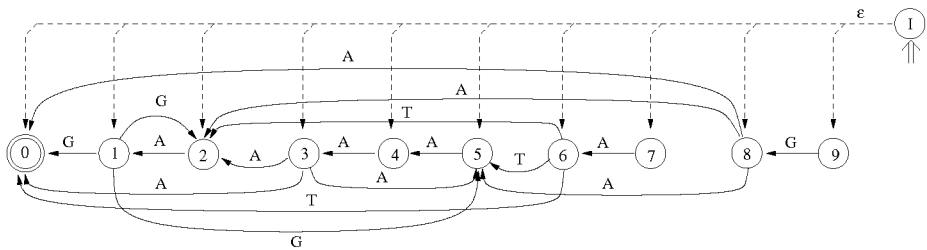


Fig. 5.24. Automaton to recognize all the reverse prefixes of the regular expression $((GA|AAA)^*(TA|AG))$.

which case we shift the window and restart the process, or because we reach the beginning of the window.

In the latter case, as for extended patterns (Chapter 4), we cannot guarantee an occurrence of the regular expression, just a factor of it. So, if the final state of the automaton has been reached at the beginning of the window, we start a forward verification using the normal DFA without an initial self-loop.

The above scheme can be improved. If we are at window position $j \leq \ell_{min}$, it is not relevant whether an automaton state at a distance greater than j from the initial state 0 is still active, because that state can never activate state 0 within the window. So we keep masks $Reach_j$ for $j \in 0 \dots \ell_{min}$, which contain the states that can influence the final result from window position j . By removing active states that are not in $Reach_j$, we are able to shift the window sooner.

Figure 5.24 makes it clear that the Glushkov property of all the arrows arriving at a given state being labeled by the same character does not hold when we reverse the arrows. Therefore, the **BP**Glushkov simulation cannot be applied directly. However, we can obtain a similar result by noticing that a dual property holds after we reverse the arrows: All the arrows leaving a given state are labeled by the same character.

Therefore we can use again tables T_d and B as before, but this time we have to mask with B before using T_d . That is, we keep the active states whose arrows leave by the current character and then take all the transitions leaving them. Formally, $\delta(D, \sigma) = T_d[D \ \& \ B[\sigma]]$, where B corresponds to the forward transitions.

Figure 5.25 shows the preprocessing algorithm, which yields a forward automaton (B, Tf_d) , a backward automaton Tb_d , and the table $Reach$. Table Tf_d is obtained by making the input NFA deterministic without adding a

```

Compute_Reach ( $T_d, I, \ell min$ )
1.   $Reach_0 \leftarrow I$  /* the initial state */
2.  For  $j \in 1 \dots \ell min$  Do
3.     $Reach_j \leftarrow Reach_{j-1} \mid T_d[Reach_{j-1}]$ 
4.  Return  $Reach$ 

Reverse_Arrows ( $N = (Q_n, \Sigma, I_n, F_n, Bf_n)$ )
5.  For  $i \in 0 \dots m, \sigma \in \Sigma$  Do
6.     $Bb_n[i, \sigma] \leftarrow 0^{m+1}$ 
7.    For  $j \in 0 \dots m$  Do
8.      If  $Bf_n[j, \sigma] \& 0^{m-i}10^i \neq 0^{m+1}$  Then
9.         $Bb_n[i, \sigma] \leftarrow Bb_n[i, \sigma] \mid 0^{m-j}10^j$ 
10.     End of if
11.   End of for
12. End of for
13. Return  $Bb_n$ 

BNDM_Preproc ( $N = (Q_n, \Sigma, I_n, F_n, B_n), \ell min$ )
/* ( $B, Tf_d$ ) (no initial self-loop) is used for verification */
14. ( $B, Tf_d$ )  $\leftarrow$  BuildTran( $N$ )
/* Reach tells reachable states */
15.  $Reach \leftarrow$  Compute_Reach( $Tf_d, I_n, \ell min$ )
/*  $Tb_d$  is a DFA for recognizing reverse prefixes */
16.  $Bb_n \leftarrow$  Reverse_Arrows( $N$ )
17. ( $Bb, Tb_d$ )  $\leftarrow$  BuildTran( $Nb = (Q_n, \Sigma, 1^{m+1}, 0^m1, Bb_n)$ )
18. Return ( $B, Tf_d, Tb_d, Reach$ )

```

Fig. 5.25. Preprocessing for the BNDM-based algorithm.

self-loop at the initial state. $Reach$ is obtained by starting at the initial state of (B, Tf_d) and performing up to ℓmin transitions by any character. Finally, Tb_d is obtained by reversing all the arrows of the NFA and then making it deterministic. The overall process takes time $O(2^m + m^2|\Sigma|)$. Figure 5.26 shows the search algorithm.

An extra space improvement is possible: Since we are interested only in the states that can be reached in at most ℓmin steps from state 0, it is not necessary to use the whole automaton with the reverse arrows; only the states belonging to $Reach_{\ell min}$ are relevant. By discarding the others we can save space.

Since at window position j we remove the states that cannot reach state 0, we keep a given state active only if it can become a prefix of length ℓmin of an occurrence. Hence, the algorithm is just another mechanism to search for *Pref* (Section 5.5.1). However, it uses the same automaton with the arrows reversed to represent the state of the search instead of the full trie as in the **MultiStringRE** algorithm.

```

RegularBNDM( $N = (Q, \Sigma, I, F, \Delta)$ ,  $\ell_{min}$ )
1.  Preprocessing
2.    ( $B, T_{fd}, T_{bd}, Reach$ )  $\leftarrow$  BNDM.Preproc( $N, \ell_{min}$ )
3.  Searching
4.     $pos \leftarrow 0$ 
5.    While  $pos \leq n - \ell_{min}$  Do
6.       $j \leftarrow \ell_{min}, last \leftarrow \ell_{min}$ 
7.       $D \leftarrow Reach_{\ell_{min}}$ 
8.      While  $D \neq 0^{m+1}$  AND  $j > 0$  Do
9.         $D \leftarrow T_{bd}[D \ \& \ B[t_{pos+j}]] \ \& \ Reach_{j-1}$ 
10.        $j \leftarrow j - 1$ 
11.       If  $D \ \& \ 0^m 1 \neq 0^{m+1}$  Then /* prefix recognized */
12.         If  $j > 0$  Then  $last \leftarrow j$ 
13.       Else /* check a possible occurrence starting at  $pos + 1$  */
14.          $D \leftarrow 0^m 1, j \leftarrow pos + 1$ 
15.         While  $j \leq n$  AND  $D \ \& \ F_n = 0^{m+1}$  AND  $D \neq 0^{m+1}$  Do
16.            $D \leftarrow T_{fd}[D] \ \& \ B[t_j]$ 
17.         End of while
18.         If  $D \ \& \ F_n \neq 0^{m+1}$  Then
19.           Report an occurrence beginning at  $pos + 1$ 
20.         End of if
21.       End of if
22.     End of while
23.      $pos \leftarrow pos + last$ 
24.   End of while

```

Fig. 5.26. Extension of **BNDM** for regular expressions.

In [Nav01b] it is shown that this scheme can be improved by finding good “necessary factors” of the regular expression, just as in **MultiFactRE**. In this case the result is a subgraph of the NFA, so that any path from the initial to a final state needs to traverse the subgraph.

Example of RegularBNDM search We search for the pattern $((GA|AAA)*) (TA|AG)$ in the text AAAAGATAGAATAGAAA, marking the initial positions of occurrences. The states have been numbered according to Figure 5.19. We show the nonzero entries of Bb_n with the rows of table T_{bd} that are needed in the search. We omit the details of the forward verification.

Again, the code is slower than a simple forward scan, but this is because our particular pattern is difficult to search for in this manner.

$$Bb_n = \begin{cases} \begin{array}{|c|c|c|} \hline 1 & G & 0000100101 \\ \hline 2 & A & 0100000010 \\ \hline 3 & A & 0000100101 \\ \hline 4 & A & 0000001000 \\ \hline 5 & A & 0000010000 \\ \hline 6 & T & 0000100101 \\ \hline 7 & A & 0001000000 \\ \hline 8 & A & 0000100101 \\ \hline 9 & G & 0100000000 \\ \hline \end{array} \end{cases}$$

$$B = \begin{cases} \begin{array}{|c|c|} \hline A & 0110111100 \\ \hline C & 0000000000 \\ \hline G & 1000000010 \\ \hline T & 0001000000 \\ \hline \end{array} \end{cases}$$

$$Reach_0 = 0000000001$$

$$Reach_1 = 0101001011$$

$$Reach_2 = 1111011111$$

$$\ell_{min} = 2$$

1. AA AAGATAGAATAGAAA

$$\begin{array}{rcl} D = & 1111011111 \\ \hline \text{Reading A} & & \\ D \& B[A] = & 0110011100 \\ Tb_d = & 0101101111 \\ \& Reach_1 = & 0101001011 \\ last = & 1 \ (D \& I_n \neq 0^{m+1}) \\ \hline \text{Reading A} & & \\ D \& B[A] = & 0100001000 \\ Tb_d = & 0000100101 \\ \& Reach_0 = & 0000000001 \end{array}$$

$D \& I_n \neq 0^{m+1}$, so we start a verification at position 1. After 5 steps we find the pattern and report it. Then we shift the window by $last = 1$.

2. A AA AGATAGAATAGAAA

As for Step 1, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 2. After 7 steps we find the pattern and report it. Then we shift the window by $last = 1$.

3. AA AA GATAGAATAGAAA

As for Step 1, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 3. After 3 steps the automaton runs out of active states, so we discard position 3 and shift by $last = 1$.

4. AAA AG ATAGAATAGAAA

$$\begin{array}{rcl} D = & 1111011111 \\ \hline \text{Reading G} & & \\ D \& B[G] = & 1000000010 \\ Tb_d = & 0100100101 \\ \& Reach_1 = & 0100000001 \\ last = & 1 \ (D \& I_n \neq 0^{m+1}) \end{array}$$

$$\begin{array}{rcl} D = & 0100000001 \\ \hline \text{Reading A} & & \\ D \& B[A] = & 0100000000 \\ Tb_d = & 0000100101 \\ \& Reach_0 = & 0000000001 \end{array}$$

$D \& I_n \neq 0^{m+1}$, so we start a verification at position 4. After 2 steps we find the pattern and report it. Then we shift the window by $last = 1$.

5. AAAA GA TAGAATAGAAA

$$\begin{array}{rcl} D = & 1111011111 \\ \hline \text{Reading A} & & \\ D \& B[A] = & 0110011100 \\ Tb_d = & 0101101111 \\ \& Reach_1 = & 0101001011 \\ last = & 1 \ (D \& I_n \neq 0^{m+1}) \\ \hline \text{Reading G} & & \\ D \& B[G] = & 0000000010 \\ Tb_d = & 0000100101 \\ \& Reach_0 = & 0000000001 \end{array}$$

$D \& I_n \neq 0^{m+1}$, so we start a verification at position 5. After 4 steps we find the pattern and report it. Then we shift the window by $last = 1$.

6. AAAAG AT AGAATAGAAA

$$\begin{array}{rcl} D = & 1111011111 \\ \hline \text{Reading T} & & \\ D \& B[T] = & 0001000000 \\ Tb_d = & 0000100101 \\ \& Reach_1 = & 0000000001 \\ last = & 1 \ (D \& I_n \neq 0^{m+1}) \\ \hline \text{Reading A} & & \\ D \& B[A] = & 0000000000 \\ Tb_d = & 0000000000 \\ \& Reach_0 = & 0000000000 \end{array}$$

$D \& I_n = 0^{m+1}$, so we shift by $last = 1$.

7. AAAAGA TA GAATAGAAA

$D =$	1 1 1 1 0 1 1 1 1 1
Reading A	
$D \& B[A] =$	0 1 1 0 0 1 1 1 0 0
$Tb_d =$	0 1 0 1 1 0 1 1 1 1
$\& Reach_1 =$	0 1 0 1 0 0 1 0 1 1
$last =$	1 ($D \& I_n \neq 0^{m+1}$)
Reading T	
$D \& B[T] =$	0 0 0 1 0 0 0 0 0 0
$Tb_d =$	0 0 0 0 1 0 0 1 0 1
$\& Reach_0 =$	0 0 0 0 0 0 0 0 0 1

$D \& I_n \neq 0^{m+1}$, so we start a verification at position 7. After 2 steps we find the pattern and report it. Then we shift the window by $last = 1$.

8. AAAAGAT AG AATAGAAA

As for Step 4, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 8. After 2 steps we find the pattern and report it. Then we shift the window by $last = 1$.

9. AAAAGATA GA ATAGAAA

As for Step 5, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 9. After 4 steps the automaton runs out of active states and we shift the window by $last = 1$.

10. AAAAGATAG AA TAGAAA

As for Step 1, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 10. After 3 steps the automaton runs out of active states, so we shift by $last = 1$.

11. AAAAGATAGA AT AGAAA

As for Step 6, $D \& I_n = 0^{m+1}$, so we shift by $last = 1$.

12. AAAAGATAGAA TA GAAA

As for Step 7, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 12. After 2 steps we find the pattern and report it. Then we shift the window by $last = 1$.

13. AAAAGATAGAAT AG AAA

As for Step 4, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 13. After 2 steps we find the pattern and report it. Then we shift the window by $last = 1$.

14. AAAAGATAGAATA GA AA

As for Step 5, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 14. After 4 steps the text finishes without recognizing the pattern, so we shift the window by $last = 1$.

15. AAAAGATAGAATAG AA A

As for Step 1, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 15. After 3 steps the text finishes without recognizing the pattern, so we shift the window by $last = 1$.

16. AAAAGATAGAATAGA AA

As for Step 1, $D \& I_n \neq 0^{m+1}$, so we start a verification at position 16. After 2 steps the text finishes without recognizing the pattern, so we shift the window by $last = 1$.

5.6 Experimental map

Determining the best search algorithm for a regular expression is more difficult than for simple patterns, because the structure of the regular expression plays a complex role in the efficiency.

An obvious disadvantage of the bit-parallel versions compared to **DFA-Classical** is that the bit-parallel algorithms build all the $2^{|Q|}$ possible combinations, while **DFAClassical** builds only the reachable states. Thus **DFA-Classical** may produce a much smaller automaton.

On the other hand, there are important advantages to the bit-parallel versions. One is that they are simpler to code. Another is that they are more flexible. For example, we will see in Chapter 6 that this scheme can be extended to permit differences between the pattern and its occurrences, which is hard to do with **DFAClassical**. Finally, bit-parallel versions are amenable to horizontal partitioning, which permits reducing the space as much as necessary.

Among bit-parallel versions, **BPGlushkov** is preferable to **BPTompson** because it needs less space and has more locality of reference as it addresses a smaller table.

Finally, **NFAModules** obtains the same space-dependent complexity as **BPGlushkov**, $O(mn/\log s)$, but it is more complicated to implement and slower in practice. However, when the regular expression needs more than, say, four or more computer words, it becomes attractive in comparison to bit-parallel algorithms. Moreover, **NFAModules** can also be extended to handle classes of characters and approximate searching (Chapter 6).

Filtration approaches, depending on the regular expression structure, can be better or worse than the previous approaches. It is difficult to define a parameter that always works well at predicting the behavior of filtration, but a good approximation is

$$Prob-verif = \frac{|Pref|}{|\Sigma|^{\ell_{min}}}$$

which is an approximation of the probability of matching a string in $Pref$, defined in Section 5.5.1. Each time an element in $Pref$ matches, we have to perform a verification whose cost is difficult to bound, but on average it can be approximated by

$$Cost-verif = \sum_{\ell \geq 0} \frac{|Pref_{\ell}|}{|\Sigma|^{\ell}}$$

where $Pref_{\ell}$ is the set of all prefixes of length ℓ of possible occurrences of

the regular expression. Those sets can be obtained with the same algorithm that computes *Pref* (Section 5.5.1).

A general rule of thumb is that filtration should be used only when

$$Cost-filter = Prob-verif \times Cost-verif \leq 1$$

The value *Pref* used works well with **MultiStringRE** and **RegularBNDM** based approaches, but for **MultiFactRE** it must be changed to the set of strings chosen there.

With respect to the different filtration approaches, **MultiStringRE** and **RegularBNDM** are similar in terms of text characters considered, especially if **Multiple BNDM** or **SBDM** is used for **MultiStringRE** (Section 3.4). **RegularBNDM** uses a compact representation of the set *Pref* by cleverly using the automaton itself instead of a fully developed trie of all alternatives. But, when the regular expression is too large **RegularBNDM** takes too much time and it is a good idea to resort to **MultiStringRE**. Another advantage of **MultiStringRE** is that it does not need to re-read the window.

Finally, **MultiFactRE**-like filtration can be seen as an improvement over the previous approaches. In particular, *Gnu Grep* (Section 7.1.1) works better than the plain **MultiStringRE** approach, and *Nrgrep* (Section 7.1.3) contains an implementation of **RegularBNDM** that also finds the best necessary factor of the regular expression.

Even for small patterns it sometimes happens that *Gnu Grep* is faster than *Nrgrep*, but **RegularBNDM** can be extended to approximate searching, while search algorithms based on classical multipattern matching normally cannot.

Table 5.2 summarizes our recommendations.

	Low <i>Cost-filter</i> (below 1.0)	High <i>Cost-filter</i> (above 1.0)
Small size ($m \leq 4w$)	RegularBNDM / MultiFactRE	DFAClassical / BPGlushkov
Large size ($m > 4w$)	MultiFactRE	NFAModules

Table 5.2. *The algorithms we recommend to search for a regular expression according to some parameters of the pattern.*

5.7 Other algorithms and references

NFA construction A theoretic lower bound to the number of transitions needed to build an ε -free NFA is $O(m \log m)$ [HSW97]. Reaching the lower bound is still an open issue. An $O(m^2)$ time algorithm producing an NFA with $O(m \log^2 m)$ transitions was proposed in [HSW97]. It was improved to $O(m \log^2 m)$ time in [HM98]. Unfortunately, this algorithm is too complicated for our purposes.

Set of regular expressions A natural extension of the regular expression search problem is that of searching for a set of regular expressions RE_1, RE_2, \dots, RE_r . In principle, this can be converted into the basic single-pattern problem by searching for $RE_1 \mid RE_2 \mid \dots \mid RE_r$. However, many of the algorithms presented do not work well with very large expressions.

One algorithm that is able to deal with large expressions is **NFAModules**, but its cost grows linearly with the size of the pattern, in our case, with r . Much better algorithms are **MultiStringRE** and **MultiFactRE**, provided the expressions can be searched for efficiently by filtration algorithms.

5.8 Building a parse tree

We show in this section how to parse a regular expression to obtain its parse tree, which in general is not unique. In the tree, each leaf is labeled by a character of $\Sigma \cup \{\varepsilon\}$ and each internal node by an operator in the set $\{|\cdot, *, \cdot\}$.

The general approach is to consider a regular expression as a string generated by a grammar, and then use the classical Unix tools *Lex* and *Yacc* or Gnu *Flex* and *Bison* to generate from the grammar the automaton that recognizes the regular expression and transforms it into a tree. The theory behind these tools can be found in books on compilers, such as [ASU86].

This general approach is valuable for large grammars, for instance, for parsers of programming languages, and for very simple grammars that need just lexical analyzers like *Lex* or *Flex*. The grammar for regular expressions is too complex to be addressed by a lexical analyzer and too simple to deserve a full bottom-up parser. The best approach is to build a simple parser by hand, and this is what we do in Figure 5.27. It assumes that the regular expression is well written and that it terminates with a special character '\$'. It also assumes that the concatenation operator “.” is implicit. Of course, this simple parser has to be modified to handle various types of errors when used in a real application, but this pseudo-code should be a useful starting point, and enough for simple applications.

```

Parse( $p = p_1 p_2 \dots p_m, last$ )
1.   $v \leftarrow \theta$ 
2.  While  $p_{last} \neq \$$  Do
3.      If  $p_{last} \in \Sigma$  OR  $p_{last} = \varepsilon$  Then      /* normal character */
4.           $v_r \leftarrow$  Create a node with  $p_{last}$ 
5.          If  $v \neq \theta$  Then  $v \leftarrow \boxed{\cdot} (v, v_r)$ 
6.          Else  $v \leftarrow v_r$ 
7.           $last \leftarrow last + 1$ 
8.      Else If  $p_{last} = '|'$  Then                      /* union operator */
9.           $(v_r, last) \leftarrow \text{Parse}(p, last + 1)$ 
10.          $v \leftarrow [ \ ] (v, v_r)$ 
11.      Else If  $p_{last} = '*'$  Then                      /* star operator */
12.           $v \leftarrow \boxed{*} (v)$ 
13.           $last \leftarrow last + 1$ 
14.      Else If  $p_{last} = '('$  Then                      /* open parenthesis */
15.           $(v_r, last) \leftarrow \text{Parse}(p, last + 1)$ 
16.           $last \leftarrow last + 1$ 
17.          If  $v \neq \theta$  Then  $v \leftarrow \boxed{\cdot} (v, v_r)$ 
18.          Else  $v \leftarrow v_r$ 
19.      Else If  $p_{last} = ')'$  Then                      /* close parenthesis */
20.          Return  $(v, last)$ 
21.      End of if
22.  End of while
23.  Return  $(v, last)$ 

```

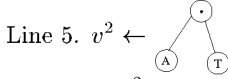
Fig. 5.27. A basic recursive parser for a well-written regular expression. θ is the empty tree.

Instead of explaining in depth how this parser works, we show its behavior on our regular expression $(AT|GA)((AG|AAA)*)$.

Parsing example We parse the regular expression $(AT|GA)((AG|AAA)*)$ using $\text{Parse}((AT|GA)((AG|AAA)*)\$, 1)$. We number the recursive calls using Parse^1 , Parse^2 , and so on. The corresponding variables are marked the same way.

- | | |
|---|--|
| <p>1. $\text{Parse}^1((AT GA)((AG AAA)*) , 1)$
 $last^1 = 1, v^1 = \theta$,
 we read $\boxed{(}$ $AT GA)((AG AAA)*)\\$.
 Line 15. We call:</p> | <p>2. $\text{Parse}^2((AT GA)((AG AAA)*) , 2)$
 $last^2 = 2, v^2 = \theta$,
 we read \boxed{A} $T GA)((AG AAA)*)\\$.
 Line 4. $v_r^2 \leftarrow \textcircled{A}$
 Line 6. $v^2 \leftarrow v_r^2$.
 Line 7. $last^2 = 3$.
 We return to the while loop of Parse^2,
 line 2.</p> |
|---|--|

3. As $p_{last^2} \neq \$$,
we read $(A \boxed{T} | GA) ((AG|AAA)*)\$$.
Line 4. $v_r^2 \leftarrow \boxed{T}$

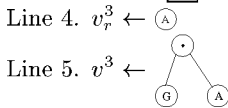


Line 7. $last^2 = 4$.

4. We enter line 8,
we read $(AT [|] GA) ((AG|AAA)*)\$$.
Line 9. We call:

5. **Parse**³((AT|GA)((AG|AAA)*), 5)
 $last^3 = 5$, $v^3 = \theta$,
we read $(AT| \boxed{G} A) ((AG|AAA)*)\$$.
Line 4. $v_r^3 \leftarrow \boxed{G}$
Line 6. $v^3 \leftarrow v_r^3$.
Line 7. $last^3 = 6$.
We return to the while loop of **Parse**³,
line 2.

6. As $p_{last^3} \neq \$$,
we read $(AT|G \boxed{A}) ((AG|AAA)*)\$$.

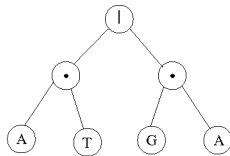


Line 7. $last^3 = 7$.

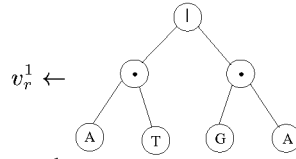
7. We enter line 19,
we read $(AT|GA \boxed{ }) ((AG|AAA)*)\$$.
Line 20. We quit the function **Parse**³.
We return $(v^3, 7)$.
Coming back to **Parse**² line 9,



Line 10. $v^2 \leftarrow$



8. We enter line 19,
we read again $(AT|GA \boxed{ }) ((AG|AAA)*)\$$.
Line 20. We quit the function **Parse**².
We return $(v^2, 7)$.
Coming back to **Parse**¹ line 15,



$last^1 \leftarrow 7$.

Line 16. $last^1 \leftarrow 8$.

Line 18. $v^1 \leftarrow v_r^1$.

We return to the while loop of **Parse**¹,
line 2.

9. As $p_{last^1} \neq \$$,
we read $(AT|GA) (\boxed{ (} (AG|AAA)*)\$$.
Line 15. We call:

10. **Parse**²((AT|GA)((AG|AAA)*), 9)
 $last^2 = 9$, $v^2 = \theta$.
As $p_{last^2} \neq \$$,
we read $(AT|GA) (\boxed{ (} \boxed{ (} AG|AAA)*)\$$.
Line 15. We call:

11. **Parse**³((AT|GA)((AG|AAA)*), 10)
 $last^3 = 10$, $v^3 = \theta$,

we read $(AT|GA) ((\boxed{ A } G|AAA)*)\$$.

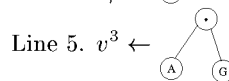
Line 4. $v_r^3 \leftarrow \boxed{A}$

Line 6. $v^3 \leftarrow v_r^3$.

Line 7. $last^3 = 11$.

We return to the while loop of **Parse**³,
line 2.


12. As $p_{last^3} \neq \$$,
we read $(AT|GA) ((A \boxed{ G } |AAA)*)\$$.
Line 4. $v_r^3 \leftarrow \boxed{G}$




Line 7. $last^3 = 12$.

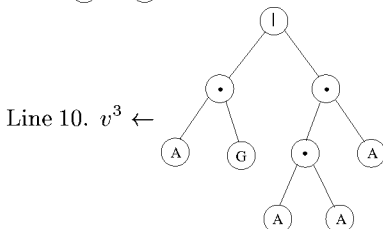
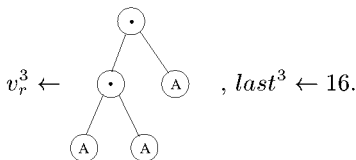
13. We enter line 8,
we read $(AT|GA) ((AG [|] AAA)*)\$$.
Line 9. We call:

14. **Parse**⁴((AT|GA)((AG|AAA)*), 13)
 $last^4 = 13, v^4 = \theta$,
 we read (AT|GA)((AG| A AA)*).
 Line 4. $v_r^4 \leftarrow \textcircled{A}$
 Line 6. $v^4 \leftarrow v_r^4$.
 Line 7. $last^4 = 14$.
 We return to the while loop of **Parse**⁴,
 line 2.

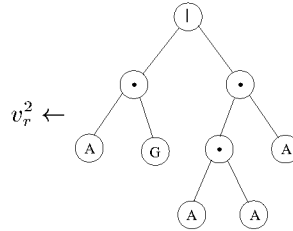
15. As $p_{last^4} \neq \$$,
 we read (AT|GA)((AG|A A A)*).
 Line 4. $v_r^4 \leftarrow \textcircled{A}$

 Line 5. $v^4 \leftarrow$
 Line 7. $last^4 = 15$.
 We return to the while loop of **Parse**⁴,
 line 2.

16. As $p_{last^4} \neq \$$,
 we read (AT|GA)((AG|AA A)*)\$.
 Line 4. $v_r^4 \leftarrow \textcircled{A}$

 Line 5. $v^4 \leftarrow$
 Line 7. $last^4 = 16$.

17. We enter line 19,
 we read (AT|GA)((AG|AAA))*\$.
 Line 20. We quit the function **Parse**⁴.
 We return ($v^4, 16$).
 Coming back to **Parse**³ line 9,

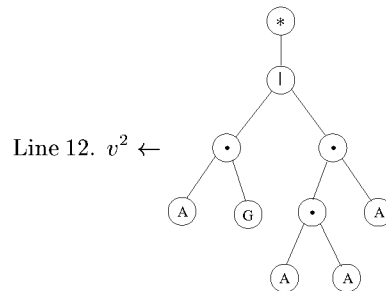


18. We enter line 19,
 we read again (AT|GA)((AG|AAA))*\$.
 Line 20. We quit the function **Parse**³.
 We return ($v^3, 16$).
 Coming back to **Parse**² line 15,



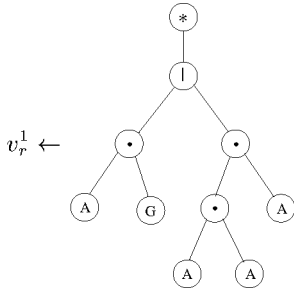
- $last^2 \leftarrow 16$
 Line 16. $last^2 \leftarrow 17$.
 Line 18. $v^2 \leftarrow v_r^2$.
 We return to the while loop of **Parse**²,
 line 2.

19. As $p_{last^2} \neq \$$,
 we read (AT|GA)((AG|AAA *)*)\$.



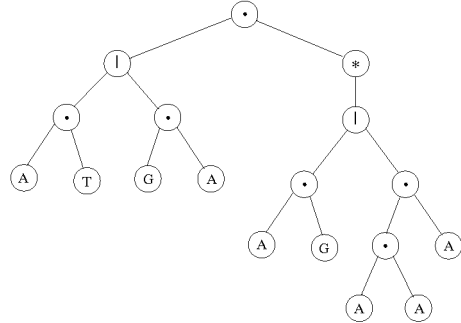
- Line 13. $last^2 \leftarrow 18$.

20. We enter line 19,
 we read $(AT|GA)((AG|AAA)* \boxed{}) \$$.
 Line 20. We quit the function **Parse**².
 We return $(v^2, 18)$.
 Coming back to **Parse**¹ line 15,



$last^1 \leftarrow 18$
 Line 16. $last^1 \leftarrow 19$.

Line 17. $v^1 \leftarrow$



We return to the while loop of **Parse**¹,
 line 2.

21. As $p_{last^1} = \$$,
 We stop the function and return
 $(v^1, 19)$.

