Generation of Pattern-Matching Algorithms by Extended Regular Expressions

Ikuo NAKATA *

Summary. It is difficult to express the definition of the comments of C language in a regular expression. However, the definition can be expressed by a simple regular expression by introducing a special symbol, called the *any*-symbol, that represents *any single character*, or by introducing a kind of negation symbol into regular expressions. In general, the problem of string pattern matching can be expressed as such an extended regular expression, and the corresponding finite state automaton generated from the expression is equivalent to the Knuth-Morris-Pratt pattern-matching algorithm [4]. In particular, if we use the *any*-symbols, the pattern is not restricted to a string of characters. It can be any regular expression. Our method can also be applied to the problem of repeated pattern matching. The Aho-Corasick algorithm [3] can be derived mechanically from an extended regular expression that contains *any*-symbols.

1 Introduction

The definition of a comment of C language can be given as a string composed of three strings in the following order:

`/*` ,
a string that does not contain `*/` (may be an empty string),
`*/` .

Expressing the above definition by a regular expression is complicated because it is difficult to express "a string that does not contain `*/`." Here, `*/` means a string of length two composed of `*` followed by `/`.

Although we can use the negation symbols in *lex*, a well-known scanner generator, the expression for the comment is still complex–for example,

```
/*/*([^*/]|[^*]/|*[^/])***/          (1)
```

in [6], or

```
/*([^*]|**[^*/])***/                (2)
```

in [5]. Here, (,),|,*,[,],^ are meta symbols, * denotes the repetition of the immediate predecessor zero or more times, and `[^*/]` expresses one character that is neither `*` nor `/`.

This problem can be simply expressed if we can designate "a string that does not contain `*/`" directly. If the expression of the above string is

```
^(*/)
```

* Institute of Information Sciences and Electronics, University of Tsukuba.

Copyright © 1996by Academic Press, Inc. and Iwanami Shoten, Publishers.
All rights of reproduction in any form reserved.
ISBN 0–12–037103–0

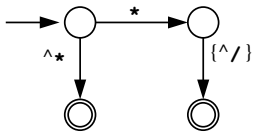

Fig. 1 The automaton to recognize ^(*/)

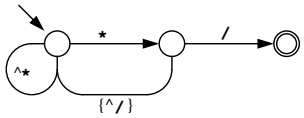

Fig. 2 DFA for (^(*/))*/

then the definition of the comment is

/*(^(*/))**/.

This is the direct expression of the definition. Here, we define the meaning of ^(*/) to be "the head character of a string that does not contain */." In Figure 1, this definition is shown as an automaton that recognizes ^(*/).

In Figure 1, ^* denotes the transition by any character except *, and {^/} indicates the look-ahead transition by ^/ (that is, the transition by looking at any character except / without reading the character). The automaton for the extended regular expression

(^(*/))*/

is shown in Figure 2. It is obtained by applying the well-known algorithm that generates first an NFA (Nondeterministic Finite Automaton) from a regular expression and then transforms it into a DFA (Deterministic Finite Automaton) [1, 5]. This is the automaton that recognizes the string that follows the head string /* of a comment. It is also the pattern-matching automaton that recognizes "the first string */" in a given text.

Another regular expression for "the first string */" can be

•*<*/>

where • is defined, in the context of being followed by <*/>, as any character that is not the first character of */. Namely, • in this case is the indirect expression for ^(*/). • by itself means *any single character*. We can construct the DFA for •*<*/> as follows: construct the DFA for •*(*/) first, and then eliminate all the transitions from the state that reads */. The resultant automaton corresponds to the previous regular expression (2) (see page 59 of [5]). In this paper, we generalize the above method and describe a method that mechanically generates the pattern-matching algorithm for any regular expression α (in the following, we identify an automaton with the corresponding algorithm). If α represents a fixed character string, then the generated algorithm is equivalent to the pattern-matching algorithm of Knuth-Morris-Pratt (KMP) [4]. We further show a method

for generating the algorithm of Aho-Corasick (A-C) [3], well known as an efficient algorithm that searches several patterns repeatedly.

2 The Knuth-Morris-Pratt Pattern-Matching Algorithm

2.1 Using *any*-symbols

Let Σ be an alphabet, α be a regular expression over Σ, and $[\alpha]$ be the language that α denotes. We add to Σ the special symbol $\bullet$. The meaning of $\bullet$ is defined as $[\bullet]=\Sigma$ when we ignore its context, and the context-dependent meaning is defined in the following. We introduce a form of regular expressions,

$$\bullet^*< \alpha >,$$

with the purpose of expressing the head string u of a given text x such that the tail of u is the first string that matches α. That is, $\bullet^*< \alpha >$ matches $u = vw$, where u is the shortest string such that u is the head of x, and $< \alpha >$ matches w. Here we take w as the first string if α matches both w and wz. We exclude the case $\varepsilon \in [\alpha]$, because in this case it matches every ε at any place in a given text. We define the above regular expression as follows:

Definition 1: Let $\varepsilon \notin [\alpha]$. $[\bullet^*< \alpha >] = \{\ x \in \Sigma^* \mid$ There exist y and z such that $x{=}yz, z \in [\alpha]$, and there exist no x_i such that $x{=}x_1x_2x_3, x_2 \in [\alpha], x_3 \in \Sigma^+\ \}$.

For example, if α=`ab`* and x=`cab`, then $\bullet^*< \alpha >$ matches `ca` because `b`* in α matches ε.

In the following, we consider the string pattern-matching problem. In a given text, the problem of searching the first string that is matched by α becomes the problem of recognizing the regular expression $\bullet^*< \alpha >$. We can construct the automaton for this expression with the following steps. First construct the NFA by appending the automaton for α to the one for $\bullet^*$. Next transform the NFA to the DFA by the well-known method. Then eliminate all the transitions from the states that contain the final states of the NFA for α. We will illustrate this by using the example expression `aa`*`bb`.

The automaton for $\bullet^*$ appended by the one for `aa`*`bb` is shown in Figure 3. In transforming this NFA to DFA, we follow the well-known method except for the $\bullet$-transitions. For the $\bullet$-transition from the state s to the state t in NFA, we add t to all the target states of other transitions from s in DFA, and we change the $\bullet$-transition to the transition by any character that is not contained in other transitions from s, as shown in Figure 4. For example, the target state of the `a`-transition in DFA of Figure 4 contains NFA States 1 and 2, because these are the target states of the `a`-transition and $\bullet$-transition in the NFA. Then, to make the first matched state into the final state, we eliminate all the transitions from the matched state–that is, the state that contains a final state in NFA. By applying these methods to Figure 3, we obtain Figure 5. Here, all the transitions from the state composed of NFA States 1 and 4 have been eliminated.

Figure 5 corresponds to the pattern-matching algorithm of Knuth-Morris-Pratt (KMP). However, KMP deals only with patterns of the form of $\alpha = a_1a_2 \cdots a_n, a_i \in \Sigma$. Our method can deal with general regular expressions as patterns. For example, the above `aa`*`bb` is a regular expression that contains the repeating symbol *. The

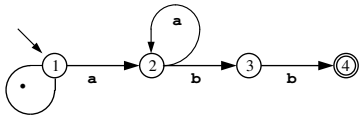

Fig. 3 NFA for •*aa*bb

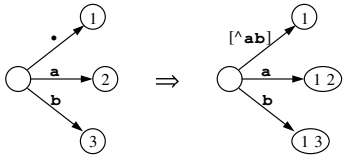

Fig. 4 Translation from NFA that contains • into DFA

corresponding automaton, the pattern-matching algorithm, can be generated by the usual method modified by the method shown in Figure 4.

2.2 Using negation symbols

• of •*< α > in the previous section can be interpreted as "the first character of strings that is different from α." To express this more directly, we introduce the notation ^α and define it as follows. The meaning of ^α is similar to the negation of α, but it depends on its context and is different from negation defined as [^α]=$\Sigma^*-[\alpha]$ in [2].

Definition 2: $[(\hat{}\alpha)\beta] = \{x \in \Sigma^* \mid$ There exist a and z such that $x{=}az, a \in \Sigma$, $z \in[\beta]$, and there exist no x_i such that $x{=}ax_1x_2, ax_1 \in[\alpha]\}$.

By using this notation, the pattern-matching problem for α can be expressed as the problem of recognizing the regular expression

$(\hat{}\alpha)^*\alpha$.

Here, by Definition 2, (^α) is interpreted under its right-hand context. Therefore (^α)(^α)α is interpreted as (^α)((^α)α). Furthermore, we exclude the case $\varepsilon \in [\alpha]$, as in the previous section. Now, as an example, we present the pattern-matching problem for `ababc`; that is, we will construct DFA for

(^(`ababc`))*`ababc`.

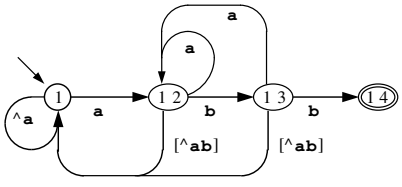

Fig. 5 DFA that recognize the first `aa*bb`

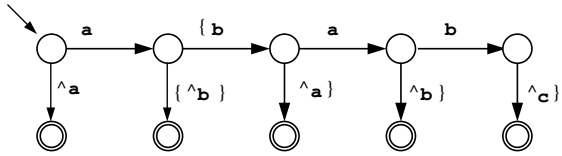

Fig. 6 DFA for ^(ababc)

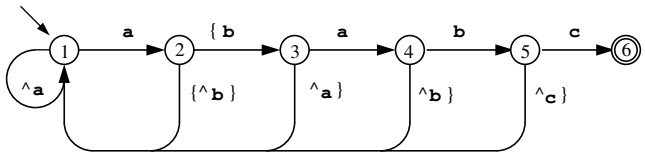

Fig. 7 DFA for (^(ababc))*ababc

DFA for ^(ababc) is shown in Figure 6.

In Figure 6, the strings composed of characters between { and } denote look-ahead strings; that is, state transitions occur by these characters as in a usual automaton, but, after recognizing ^(α), these characters are treated as if they have not been read yet. In terms of an automaton, the input tape is marked at the beginning of {a-transition, and the input tape is placed at the mark at the end of a}-transition. DFA for (^(ababc))*ababc can be obtained by first applying the usual method for NFA construction to Figure 6 and ababc, then by applying the usual algorithm for translation from NFA to DFA. In this process, the characters accompanied by { or } are treated as ordinal characters. The result shown in Figure 7 is the naïve pattern matching algorithm.

For example, if the text

abcababd

is read by the automaton of Figure 7, the sequence of state transitions is

1a2b3c1{b1c1}a2b3a4b5d1{b1a2b3d1}{b1d1},

and the recognition fails. Here, the enclosing braces mean "reading again." In this algorithm, after ab^a (abc in this example) has been read, b^a is read again, as shown in a{b^a}. This is equivalent to reading two ^as at the initial state. The resultant state is the initial state as shown in 1{b1c1}. Therefore, in this case, it is equivalent to returning to the initial state without being read again. In other words, { and } can be eliminated from the automaton. By applying similar optimization to other transitions, DFA in Figure 8 is obtained. When ^c is read at State 5 of Figure 7, the transition proceeds strictly as 1{b1a2b3}, then the ^c that is last read should be read again to see whether it is a or not. This transition is shown in the transition from State 5 to State 3 in Figure 8. The algorithm of Figure 8 is equivalent to the pattern-matching algorithm of KMP (that is shown in the first part of [4] and is referred in many text books of algorithms as KMP's pattern-matching algorithm).

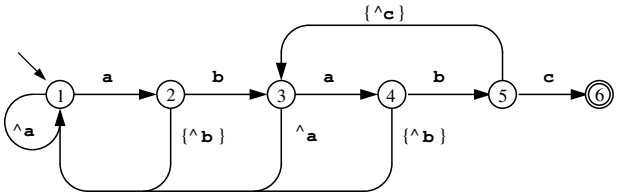

Fig. 8 DFA for (^(ababc))*ababc(after optimization)

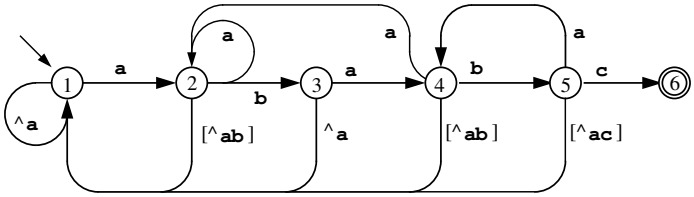

Fig. 9 DFA for •*< ababc >

Note that this optimization cannot be applied, in general, to the automaton that contains loops, because the string that should be read again cannot be determined uniquely. aa*bb in the previous section is such an example.

Figure 9 is obtained by applying the method in the previous section that uses *any*-symbols to the above example ababc. This is the optimized version of Figure 8, because this does not contain any read-again transitions. This is equivalent to the algorithm written in the last part of [4].

3 The Aho-Corasick Algorithm

The Aho-Corasick algorithm is well known as an efficient algorithm for searching several patterns simultaneously and repeatedly. This algorithm can be derived by a method similar to the one in the previous section.

3.1 Using *any*-symbols

In a given text x, in order to search all the strings that are matched by the patterns of α, it is sufficient to search for all the strings in the head part of x that are matched by α first, then to search for $\bullet\alpha$ in the head of x, and then to search for $\bullet\bullet\alpha$ in the head of x, and so on. Therefore this problem is equivalent to the problem of pattern matching with

$$\bullet^*\alpha.$$

The algorithm can be derived as DFA for $\bullet^*\alpha$ in which the state that contains the final states of α records the facts that it recognized α.

For example, let

$$\alpha = (\texttt{he}|\texttt{she}|\texttt{his}|\texttt{hers}).$$

This is the example in [3]. NFA for $\bullet^*\alpha$ is shown in Figure 10.

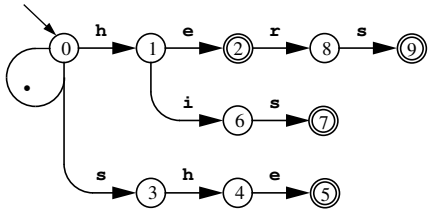

Fig. 10 NFA for •*(he|she|his|hers)

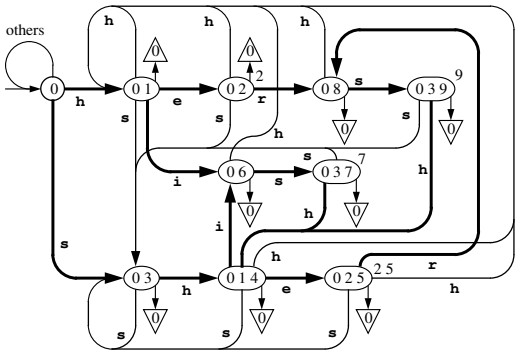

Fig. 11 DFA for repeated recognition of (he|she|his|hers)

DFA shown in Figure 11 can be obtained from this NFA by the well-known method modified by •-transitions, as shown in Figure 4. In Figure 11, every state contains one or more states in the original NFA, and numbers on the right side of a state are the numbers of the final states in NFA. The numbers 2, 5, 7, and 9 indicate that they are the states that record the recognition of `he`, `she`, `his`, and `hers`, respectively. An arrow to the number 0 in a triangle indicates the transition to the State 0 in the case of "other characters." For example, a state transition from State (0 1) to State 0 occurs in case of characters other than `h`, `s`, `e`, `i`. Although Figure 11 looks complex, it is not difficult to understand. For example, the transitions from State (0 2 5) are the same as the transitions from State (0 2) because there is no transition from State 5 in NFA. Similarly, the transitions from State (0 3 7) and from State (0 3 9) are the same as the ones from State (0 3). Every state of this DFA contains State 0 of NFA and the corresponding transitions. The rest of DFA that is shown in bold arrows in Figure 11 looks similar to Figure 10.

The algorithm in Figure 11 is the same as the final optimized algorithm in [3]. Here it is derived mechanically by slightly expanding the well-known method.

3.2 Using negation symbols

$\hat{}\alpha$ was introduced to indicate the head character of a string that is not matched by α. Here, we introduce one more symbol to indicate the head character of a string

that is matched by α:

$$^{\circ}\alpha.$$

By using this symbol, the algorithm for searching all strings that are matched by α in a given text can be derived from the following regular expression:

$$(\hat{}\alpha|^{\circ}\alpha)^*.$$

We need to add to the derived automaton the actions of recording the recognition of α that is found when $^{\circ}\alpha$ is recognized. By applying a similar method to the ones in Section 2.2 to this $(\hat{}\alpha|^{\circ}\alpha)^*$, we can obtain an algorithm that is equivalent to the ones in [3]. However, we do not explain the method here, because it is not as simple as the previous method.

4 Conclusion

We introduced the symbol $\bullet$ for representing *any single character* in regular expressions, and by giving a context dependent interpretation to $\bullet^*$ we showed that the definition of the comments in C language, which is difficult to define by a regular expression, can be expressed naturally by a simple expression. By applying this method to general regular expressions, the pattern-matching algorithm of Knuth-Morris-Pratt can be derived mechanically not only for the pattern of a character string but for the pattern expressed by a general regular expression. We also showed that a similar process is possible by introducing a symbol that indicates a kind of negation expressions.

Furthermore, we showed that the problem of repeated pattern matching for several patterns can be expressed in a simple regular expression by using $\bullet^*$, and the Aho-Corasick algorithm can be derived mechanically from the expression.

Although we did not invent any essentially new pattern-matching algorithm by these methods, we succeeded in generalizing the pattern expressions. Because the problems of pattern matching can be expressed in regular expressions, our methods can be implemented in lexical analyzer generators such as *lex*.

Acknowledgments. We would like to express our thanks to Prof. Masataka Sassa, Tokyo Institute of Technology, for his valuable comments.

References

[1] Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] Aho, A.V., Sethi, R. and Ullman, J.D.: *Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] Aho, A.V. and Corasick, M.J.: Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM*, Vol. 18, No. 6 (1975), pp. 333-340.

[4] Knuth, D.E., Morris, J.H. Jr. and Pratt, V.R.: Fast Pattern Matching in Strings, *SIAM J. Comput.*, Vol. 6, No. 2 (1977), pp. 323-350.

[5] Nakata, I.: *Compiler*, Sangyo-Tosho, Tokyo, 1981 (in Japanese).

[6] Schreiner, A.T. and Friedman, H.G. Jr.: *Introduction to Compiler Construction with UNIX*, Prentice-Hall, 1985.

Ikuo Nakata
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, Ibaraki
305 Japan