拡張正則表現に対する近似文字列照合問題

山本博章
信州大学工学部情報工学科

概要. 拡張正則に対する近似文字列照合問題とは、長さ m の拡張正則表現 r, 長さ n の文字列 x、任意の数 $d \geq 0$ が与えられたときに, x の接頭語 yで $edit(y, r) \leq d$ なるものすべてを見つける問題である。ここで、$edit(y, r)$ は、yと r間の最小の編集距離である。この問題を多項式時間で解くことは難しいことが知られている。本論文では, この問題を $O(N_r^2(2m)^{2N_r}n)$ 時間かつ $O(N_r^2(2m)^{N_r})$ 領域で解くための新しいアルゴリズムを示す。ここで、$N_r = 2u(r) + n(r) + 1$ で、$u(r)$ および $n(r)$ はそれぞれ、rに出現する共通集合演算子および補集合演算子の数を表す（これら二つの演算子は拡張演算子と呼ばれる)。このように、我々のアルゴリズムは拡張演算子の数に依存し、もしそれが定数ならば多項式時間で走る。

Approximate Pattern Matching Problem for Extended Regular Expressions

Hiroaki YAMAMOTO
Department of Information Engineering, Shinshu University,
4-17-1 Wakasato, Nagano-shi, 380-8553 Japan.
yamamoto@cs.shinshu-u.ac.jp

Abstract. Given an extended regular expression r of length m, a text string x of length n and any number $d \geq 0$, the approximate pattern matching problem for extended regular expressions is to find all the prefixes y of x with $edit(y, r) \leq d$. Here $edit(y, r)$ is the minimum edit distance between string y and any string generated by r. It is difficult to design a polynomial time algorithm for this problem. In this paper, we will present a new algorithm to solve the approximate pattern matching problem in $O(N_r^2(2m)^{2N_r}n)$ time and $O(N_r^2(2m)^{N_r})$ space in the worst case. Here $N_r = 2u(r) + n(r) + 1$ when $u(r)$ and $n(r)$ are the number of intersection operators and complement operators occurring in r, respectively. Thus our algorithm runs in polynomial time if $u(r)$ and $n(r)$ are constant numbers.

1 Introduction

This paper is concerned with the approximate pattern matching problem for extended regular expressions (that is, regular expressions with intersection and complement): Given an extended regular expression r of length m, a text string x of length n and any number $d \geq 0$, find all the prefixes y of x with $edit(y, r) \leq d$. Here $edit(y, r) = \min_{z \in L(r)}\{edit(y, z)\}$, where $L(r)$ denotes the set of strings (that is, language) generated by r and the edit distance $edit(y, z)$ between two strings y and z is defined to be the minimum number of insertions, deletions, and/or substitutions required to transform y into z. For example, $edit(aaabb, aabbb) = 1$ and $edit(aaabb, abbb) = 2$.

It is widely known that the classical exact pattern matching algorithm for regular expressions runs in $O(mn)$ time and $O(m)$ space, which is based on nondeterministic finite automata (NFAs for short) [1, 2, 5]. Myers [6] has improved this algorithm and has given an $O(mn/\log n)$ time and space pattern matching algorithm. Myers and Miller [7] applied the NFA-based technique to the approximate pattern matching problem of regular expressions, and showed an $O(mn)$ time and $O(m)$ space pattern matching algorithm. Wu, Manber and Myers [9] have improved this algorithm by using the similar technique to Myers [6], and showed $O(mn/\log_{d+2} n)$ time algorithm. Furthermore, Wu and Manber [8] presented another algorithm that runs fast in practice for small regular expressions. Thus the approximate pattern matching problem for regular expressions can be solved in the same time complexity as the exact pattern matching problem.

As for extended regular expressions, Hopcroft and Ullman [5] first gave the recognition algorithm based on the inductive definition of extended regular expressions, which runs in $O(mn^3)$ time and $O(mn^2)$ space. Knight and Myers [4] have given a recognition algorithm based on an extension of the inductive

transformation from regular expressions into NFAs. Their algorithm also runs in $O(mn^3)$ time and $O(mn^2)$ space in the worst case. Recently, Yamamoto [10] has given a new automata-based recognition algorithms. His algorithm runs faster than the existing one. By using these recognition algorithm, the exact pattern matching problem for extended regular expressions can be solved in the similar complexities. On the other hand, up to now, the approximate pattern matching problem has never been studied.

In this paper, we will study the approximate pattern matching problem and will present a new approximate pattern matching algorithm that is a natural extension of that in [7]. As with [7, 9], the running time is independent of d if d is a number that can be stored in one machine-word. Our algorithm runs in $O(N_r^2(2m)^{2N_r}n)$ time and $O(N_r^2(2m)^{N_r})$ space for any number $d \geq 0$ in the worst case, where $N_r = 2u(r) + n(r) + 1$ when $u(r)$ and $n(r)$ are the number of intersection operators and the number of complement operators occurring in r, respectively. Thus, the less the number of extended operators is, the faster our algorithm runs. Especially, if $u(r)$ and $n(r)$ are constant numbers then it runs in polynomial time in m and n. Furthermore, it runs $O(m^2 + mn)$ time and $O(m)$ space for regular expressions.

Our technique is an extension of the technique used in [7, 9]. They efficiently computed $E[q_j, i]$ by a dynamic programming technique, where $E[q_j, i]$ is the minimum edit distance over all $edit(a_1 \cdots a_i, y)$ such that y is a string accepted at the state q_j of the NFA obtained from a regular expression. Our extension is as follows. As in [10], we partition the parse tree of an extended regular expression into modules by extended operators, and then transform each module into an NFA called an augmented NFA. Furthermore, we introduce a new structure called a computation modular tree for such augmented NFAs, and construct a kind of NFAs based on computation modular trees. In other words, this NFA is constructed by viewing computation modular trees as states and the accepting language exactly coincides with $L(r)$. Then we are computing $E[T, i]$ for every computation modular tree T reachable from the initial computation modular tree by dynamic programming technique, where $E[T, i]$ is the minimum edit distance over all $edit(a_1 \cdots a_i, y)$ such that y is a string accepted at the computation modular tree T.

Throughout the paper, as in [1, 5, 6, 7, 8, 10], we rely on an $l = max\{\log n, \log m\}$-bit uniform RAM, that is, each l-bit instruction is executed in one unit of time and each l-bit register is stored in one unit of space, to evaluate all complexities appearing in this paper.

2 Preliminaries

Definition 1 *Let Σ be an alphabet. The extended regular expressions over Σ are defined as follows.*

1. *$\emptyset$, ϵ and a ($\in \Sigma$) are extended regular expressions that denote the empty set, the set $\{\epsilon\}$ and the set $\{a\}$, respectively.*
2. *Let r_1 and r_2 be extended regular expressions denoting the sets R_1 and R_2, respectively. Then $(r_1 \vee r_2)$, $(r_1 r_2)$, (r_1^*), $(r_1 \wedge r_2)$ and $(\neg r_1)$ are also extended regular expressions that denote the sets $R_1 \cup R_2$, $R_1 R_2$, R_1^*, $R_1 \cap R_2$, and $\overline{R_1}$ $(= \Sigma^* - R_1)$, respectively.*

By $u(r)$ and $n(r)$, we denote the number of intersection operators $\wedge$ and the number of complement operators $\neg$ occurring in an extended regular expression r, respectively. Regular expressions are defined by three operators $(r_1 \vee r_2)$, $(r_1 r_2)$ and (r_1^*), and semi-extended regular expressions are defined by four operators $(r_1 \vee r_2)$, $(r_1 r_2)$, (r_1^*) and $(r_1 \wedge r_2)$. To take advantage of hierarchical structure of extended regular expressions, we introduce their parse trees. Let r be an extended regular expression. Then the parse tree P_r is defined as follows:

1. If $r = \emptyset$ (ϵ, a, respectively), then P_r is a tree consisting of just one node labeled by $\emptyset$ (ϵ, a, respectively).
2. If $r = r_1 \vee r_2$ ($r = r_1 \wedge r_2$, $r = r_1 r_2$, $r = r_1^*$, $r = \neg r_1$, respectively), then P_r is a tree such that its root is labeled by $\vee$ ($\wedge$, $\cdot$, $*$, $\neg$, respectively) and the left subtree and the right subtree of the root are P_{r_1} and P_{r_2} ($*$ and $\neg$ have only P_{r_1}), respectively. The operator "$\cdot$" means concatenation.

The following proposition is widely known as the linear translation from regular expressions into NFAs (for example, see [5]).

Proposition 1 *Let r be a regular expression of length m. Then we can construct an NFA M such that M has at most $2m$ states and accepts the language denoted by r.*

3 Modular Tree and Augmented NFAs

As in [10], we design an algorithm by inductively constructing NFAs from hierarchical formation of extended regular expressions. Let r be an extended regular expression over an alphabet Σ and let P_r be the parse tree of r. Then, we partition P_r by nodes labeled with intersection $\wedge$ and complement $\neg$ into subtrees such that (1) the root of each subtree is either a child of a node labeled with $\wedge$ or $\neg$ in P_r or the root of P_r, (2) each subtree does not contain any interior nodes labeled by $\wedge$ or $\neg$, (3) each leaf is labeled by $\emptyset$, ϵ, $a \in \Sigma$, $\wedge$ or $\neg$. If it is labeled by $\wedge$ ($\neg$, respectively), then it is called *a universal leaf* (*a negating leaf*, respectively). These leaves are also called *a modular leaf*. We call such a subtree *a module*.

Let R and R' be modules in the parse tree P_r. If a modular leaf u of R becomes the parent of the root of R' in P_r, then R is called *a parent of* R', and conversely R' is called *a child of* R or *a child of* R *at* u. Thus there are two children at each universal leaf, while one child at each negating leaf. If the root of a module R is the root of P_r, then R is called *a root module*. If a module R does not have any children, then R is called *a leaf module*. It is clear that such a parent-child relationship induces *a modular tree* $\mathcal{T}_r = (\mathcal{R}, \mathcal{E})$ such that (1) $\mathcal{R}$ is a set of modules, (2) $(R, R') \in \mathcal{E}$ if and only if R is the parent of R'. We number the nodes of the modular tree $\mathcal{T}_r$ from 0 in the order of the root to leaves, the left to the right. By N_r, we denote the number of nodes of $\mathcal{T}_r$.

Now, for each module R, we relabel every modular leaf u of R with a new symbol σ_u called *a modular symbol*. By this relabeling, R can be viewed as a regular expression over $\Sigma \cup \{\sigma_u \mid u \text{ is a modular leaf of } R\}$. Then, by Proposition 1, we can construct an NFA M_R for a module R. We call this M_R *an augmented NFA* (A-NFA for short). This time, for a state q such that a transition $\delta(q, \sigma_u)$ is defined, q is called *a universal state* if u is a universal leaf, and q is called *a negating state* if u is a negating leaf. Note that the number of transitions from such a state q is just one if any. In addition, if a module R' is a child of R at u, then A-NFA $M_{R'}$ is said to be *associated with* σ_u or *associated with the state* q. Obviously, if a modular leaf u is universal, then two A-NFAs are associated with the modular symbol σ_u, and if it is negating, then just one A-NFA is associated with the modular symbol σ_u. The following proposition is immediately obtained from Proposition 1.

Proposition 2 *Let r be an extended regular expression of length m, let $R_0, \ldots, R_l$ be modules produced by partitioning P_r, and let m_j be the length of the subexpression of r for the module R_j. Then we can construct A-NFAs M_j for each module R_j such that the number of states of M_j is at most $2m_j$.*

4 Approximate Pattern Matching Algorithm

We first introduce a new structure called *a computation modular tree* which is an important concept to design a new approximate pattern matching algorithm for extended regular expressions.

4.1 A Computation Modular Tree

Let r be an extended regular expression over an alphabet Σ and let $x = a_1 \cdots a_n$ be a text string in Σ^*. Then assume that the parse tree P_r is partitioned into $l+1$ modules $R_0, \ldots, R_l$. Here R_0 is the root module. Let $M_0, \ldots, M_l$ be A-NFAs for each module. Then *a computation modular tree* (CMT for short) is defined as follows.

For each A-NFA M_j $(0 \leq j \leq l)$, we introduce *a state-node* U_j, which takes a state of M_j as its value. A computation modular tree $T = (\mathcal{U}, \mathcal{E})$ is defined to be a tree such that (1) $\mathcal{U}$ is the set of *nodes*, which consists of state-nodes, and $\mathcal{E}$ is the set of *edges*, which consists of pairs (U, U') of state-nodes, (2) the root is the state-node U_0 for the root A-NFA M_0, (3) let U_{j_1} and U_{j_2} be any state-nodes of $\mathcal{U}$. Then U_{j_2} is a child of U_{j_1} if and only if R_{j_2} is a child of R_{j_1} at a modular leaf u. In (3), if u is a universal leaf, then R_{j_1}

Function BuildGraph(r)
Input: an extended regular expression r;
Output: CMT transition graph $\mathcal{G} = (\mathcal{Q}, \mathcal{E})$;

Step 1. Partition P_r into modules $R_0, \ldots, R_l$.

Step 2. Translate each module R_j $(0 \leq j \leq l)$ into an A-NFA M_j.

Step 3. $\mathcal{Q} := \{T_0\}$ and $\mathcal{E} := \emptyset$, where T_0 is the initial CMT.

Step 4. $BuildCMT(\mathcal{G}, T_0)$.

Step 5. Return $\mathcal{G}$.

図 1: The function *BuildGraph*

has another child R_{j_3} at u. Hence U_{j_3} also becomes a child of U_{j_1}. This time, we call the pair (U_{j_2}, U_{j_3}) *a universal pair*. If u is a negating leaf, then R_{j_1} has just one child R_{j_2} at u. This time, we call the node U_{j_2} *a negating node*. For any state-node U, let $State(U)$ denote the state of U.

Let q_0 be the initial state of the root A-NFA M_0. Then the CMT $T_0 = (\{U_0\}, \emptyset)$ with $State(U_0) = q_0$ is called *the initial CMT*. In addition, for any CMT T, T is called *an accepting CMT* if the state of U_0 of T is the final state of M_0.

4.2 The CMT Transition Graph

The CMT transition graph for an extended regular expression is the labeled directed graph $\mathcal{G} = (\mathcal{Q}, \mathcal{E}, \Sigma, T_0, T_f)$ such that (1) $\mathcal{Q}$ is the set of CMTs reachable from T_0, (2) $\mathcal{E}$ is the set of directed edges, each of which is an ordered pair (T, T') of CMTs and labeled by a symbol in $\Sigma \cup \{\epsilon\}$, (3) Σ is an alphabet, (4) $T_0 \in \mathcal{Q}$ is the initial CMT, (5) $T_f \in \mathcal{Q}$ is the special CMT called *the final CMT*, and for any CMT $T \in \mathcal{Q}$, there is an edge (T, T_f) if and only if T is an accepting CMT.

We can view $\mathcal{G}$ as an NFA by regarding $\mathcal{Q}$ as the set of states, $\mathcal{E}$ as the transitions between states, the initial CMT as the initial state and the final CMT as the accepting state. Hence we can define the language $L(\mathcal{G})$ accepted by $\mathcal{G}$ in the standard way.

Now, to construct the CMT transition graph, we compute all CMTs reachable from the initial CMT T_0 by simulating the set of A-NFAs. This time, we must keep the following two conditions.

1. *Intersection condition*: Let M_{j_1} be an A-NFA such that M_{j_1} has a transition $\delta(q, \sigma) = p$ for a universal state q and a modular symbol σ, and let M_{j_2} and M_{j_3} be two A-NFAs associated with σ. Then the transition from state q to state p is possible if and only if M_{j_2} and M_{j_3} reach the final states at same time.

2. *Complement condition*: Let M_{j_1} be an A-NFA such that M_{j_1} has a transition $\delta(q, \sigma) = p$ for a negating state q and a modular symbol σ, and let M_{j_2} be the A-NFA associated with σ. Then the transition from state q to state p is possible if and only if M_{j_2} does not reach the final state.

Now let us describe a function *BuildGraph* that builds the CMT transition graph for a given extended regular expression. The detail is given in Fig. 1. The function *BuildGraph* first assigns only the initial CMT T_0 to $\mathcal{G}$ and then invokes the procedure *BuildCMT* given in Fig. 2. The procedure *BuildCMT* computes CMTs reachable from T_0 using the technique of depth-first search. To compute all the transitions from a CMT T, *BuildCMT* invokes *NextCMT*. A stack *Anc* is used to store all CMTs on the path from T_0 to the CMT that is currently being processed. Then all the possible transitions from T are classified into the following four cases: (a) the transition to the final CMT T_f if T is accepting, (b) transitions by ϵ-move from each state included in T, (c) transitions by a symbol $a \in \Sigma$ from every state included in T, (d) transitions by a modular symbol if T satisfies the intersection condition or the complement condition. *NextCMT* computes cases (a), (b) and (c) in 1, 2 and 3, respectively, and computes the case (d) in 4 and 5.

Let $\mathcal{G}$ be the CMT transition graph generated by *BuildGraph* for a given extended regular expression r. Then we have the following property for $\mathcal{G}$.

Procedure BuildCMT$(\mathcal{G}, T)$
$\mathcal{G} = (\mathcal{Q}, \mathcal{E})$: a CMT transition graph;
T: a CMT;

1. $mark[T] := processed$,
2. $Push(T, Anc)$, /* insert T at the top of stack Anc */
3. $Next[T] := \emptyset$,
4. $NextCMT(\mathcal{G}, T)$,
5. for all $T' \in Next[T]$ such that $mark[T'] := unprocessed$ do
   $BuildCMT(\mathcal{G}, T')$,
6. $Pop(Anc)$. /* delete the top element of stack Anc */

図 2: The Procedure *BuildCMT*

Lemma 1 *(1) The number of CMTs is at most* $2(2m+1)^{N_r}$*, (2) the number of outgoing edges from each CMT is at most* $2N_r$*, (3)* $L(r) = L(\mathcal{G})$.

The incoming edges of each state of A-NFAs are either all edges with ϵ or all edges with a symbol $a \in \Sigma$. Hence so are the incoming edges of each CMT of $\mathcal{G}$. We call such CMTs an ϵ-node and an S-node, respectively. For any CMT transition graph $\mathcal{G}$, let us define a graph $\mathcal{G}_t = (\mathcal{Q}, \mathcal{E}_t)$ to be the directed acyclic graph induced from $\mathcal{G} = (\mathcal{Q}, \mathcal{E})$ such that for any $e \in \mathcal{E}_c$, the directed graph $(\mathcal{Q}, \mathcal{E}_t \cup \{e\})$ has a cycle, where $\mathcal{E}_c = \mathcal{E} - \mathcal{E}_t$. Note that all CMTs in $\mathcal{Q}$ are topologically ordered from T_0 according to $\mathcal{G}_t$. We can compute $\mathcal{E}_c$ while constructing $\mathcal{G}$ by *NextCMT*.

4.3 The Algorithm

Let r be an extended regular expression and $x = a_1 \cdots x_n$ be a text string. Our algorithm first computes the CMT transition graph, and then for all CMTs T, computes $E[T, i]$ that is the minimum value over all $edit(a_1 \cdots a_i, y)$ such that y is a string accepted at the CMT T. Now let $Pre(T) = \{T' \mid (T', T) \in \mathcal{E}\}$ and $\overline{Pre}(T) = \{T' \mid (T', T) \in \mathcal{E}_t\}$. Furthermore, let $E[Pre(T), i] = \min_{T' \in Pre(T)} \{E[T', i]\}$ and $E[\overline{Pre}(T), i]$ is also defined similarly. Then $E[T, i]$ is defined by the following recurrence. We are computing the following recurrence for $E[T, i]$ in topological order of CMTs. As in [9], if we replace $E[T_0, i] = i$ with $E[T_0, i] = 0$ at 2-(a), then the value of $E[T, i]$ becomes the minimum edit distance over all the substrings ending at the position i of x.

1. For $i = 0$
   (a) $E[T_0, 0] = 0$,
   (b) for any CMT T such that $T \neq T_0$,
   $E^0[T, 0] = E^0[\overline{Pre}(T), 0] + \gamma$, where if T is an S-node, then $\gamma = 1$; otherwise 0,
   (c) for any CMT T such that $T \neq T_0$ and $1 \leq h \leq L$,
   $E^h[T, 0] = \min\{E^h[\overline{Pre}(T), 0], E^{h-1}[Pre(T), 0]\} + \gamma$, where if T is an S-node, then $\gamma = 1$; otherwise 0,
   (d) for any CMT T such that $T \neq T_0$, $E[T, 0] = E^L[T, 0]$.
2. For $1 \leq i \leq n$,
   (a) $E[T_0, i] = i$,
   (b) for any CMT T such that $T \neq T_0$,
      i. if T is an ϵ-node, then $E^0[T, i] = E^0[\overline{Pre}(T), i]$,
      ii. if T is an S-node, then $E^0[T, i] = \min\{E[T, i-1]+1, E[Pre(T), i-1]+\gamma, E^0[\overline{Pre}(T), i]+1\}$, where $\gamma = 0$ if a_i matches the label; otherwise $\gamma = 1$,

Procedure NextCMT$(\mathcal{G}, T)$
$\mathcal{G} = (\mathcal{Q}, \mathcal{E})$: a CMT transition graph; T: a CMT;

1. if T is an accepting CMT, then $\mathcal{E} := \mathcal{E} \cup \{(T, T_f)\}$ and label (T, T_f) with ϵ,
2. for all state-nodes U in T do
   (a) if $\delta(State(U), \epsilon)$ is defined, then for all states $p \in \delta(State(U), \epsilon)$ do
      i. copy T to T', and then change $State(U)$ to p in T' and $Next[T] := Next[T] \cup \{T'\}$,
      ii. $\mathcal{Q} := \mathcal{Q} \cup \{T'\}$, $\mathcal{E} := \mathcal{E} \cup \{(T, T')\}$, label (T, T') with ϵ, and if $T' \in Anc$ then $\mathcal{E}_c := \mathcal{E}_c \cup \{(T, T')\}$,
   (b) if $State(U)$ is a universal state, then
      i. copy T to T', an then assign $\emptyset$ to U in T',
      ii. if there is not nodes U_{j_1} and U_{j_2} in T', then generate them, where M_{j_1} and M_{j_2} are the A-NFAs associated with $State(U)$,
      iii. assign q_{j_1} and q_{j_2} to U_{j_1} and U_{j_2}, where q_{j_1} and q_{j_2} are the initial states of M_{j_1} and M_{j_2}, respectively, and $Next[T] := Next[T] \cup \{T'\}$,
      iv. $\mathcal{Q} := \mathcal{Q} \cup \{T'\}$, $\mathcal{E} := \mathcal{E} \cup \{(T, T')\}$, label (T, T') with ϵ, and if $T' \in Anc$ then $\mathcal{E}_c := \mathcal{E}_c \cup \{(T, T')\}$,
   (c) if $State(U)$ is a negating state, then
      i. copy T to T', and then assign $\emptyset$ U in T',
      ii. if there is not node U_{j_1} in T', then generate it, where M_{j_1} is the A-NFA associated with $State(U)$,
      iii. assign q_{j_1} to U_{j_1}, where q_{j_1} is the initial state of M_{j_1}, and $Next[T] := Next[T] \cup \{T'\}$,
      iv. $\mathcal{Q} := \mathcal{Q} \cup \{T'\}$, $\mathcal{E} := \mathcal{E} \cup \{(T, T')\}$, label (T, T') with ϵ, and if $T' \in Anc$ then $\mathcal{E}_c := \mathcal{E}_c \cup \{(T, T')\}$,
3. for all $a \in \Sigma$ do
   (a) copy T to T',
   (b) for all state-nodes U in T' do
      i. if $\delta(State(U), a) = p$ for a state p, then change the state of U to p,
      ii. if $\delta(State(U), a) = \emptyset$, then change the state of U to $\emptyset$,
   (c) $Next[T] := Next[T] \cup \{T'\}$, $\mathcal{Q} := \mathcal{Q} \cup \{T'\}$, $\mathcal{E} := \mathcal{E} \cup \{(T, T')\}$, label (T, T') with a symbol a, and if $T' \in Anc$ then $\mathcal{E}_c := \mathcal{E}_c \cup \{(T, T')\}$,
4. for all universal pairs (U_{j_1}, U_{j_2}) in T do /* check the intersection condition */
   (a) if U_{j_1} and U_{j_2} both have the final states, then
      i. copy T to T',
      ii. assign a state p to the parent U of U_{j_1} and U_{j_2}), where p is the state with $\delta(q, \sigma) = p$ such that the corresponding A-NFAs M_{j_1} and M_{j_2}) are associated with σ,
      iii. $Next[T] := Next[T] \cup \{T'\}$, $\mathcal{Q} := \mathcal{Q} \cup \{T'\}$, $\mathcal{E} := \mathcal{E} \cup \{(T, T')\}$, label (T, T') with ϵ, and if $T' \in Anc$ then $\mathcal{E}_c := \mathcal{E}_c \cup \{(T, T')\}$,
5. for all negating nodes U_{j_1} in T do /* check the complement condition */
   (a) if U_{j_1} does not have the final state, then
      i. copy T to T',
      ii. assign a state p to the parent U of U_{j_1}, where p is the state with $\delta(q, \sigma) = p$ such that the corresponding A-NFA M_{j_1} is associated with σ,
      iii. $Next[T] := Next[T] \cup \{T'\}$, $\mathcal{Q} := \mathcal{Q} \cup \{T'\}$, $\mathcal{E} := \mathcal{E} \cup \{(T, T')\}$, label (T, T') with ϵ, and if $T' \in Anc$ then $\mathcal{E}_c := \mathcal{E}_c \cup \{(T, T')\}$,
6. for all newly generated CMTs T', $mark[T'] := unprocessed$.

図 3: The Procedure *NextCMT*

Algorithm APPROX(r, x, d)
Input: an extended regular expression r, a text string $x = a_1 \cdots a_n$, and any number $d \geq 0$.
Output: all pairs of a position i and $edit(a_1 \cdots a_i, r)$ such that $edit(a_1 \cdots a_i, r) \leq d$.

Step 1. $\mathcal{G} :=$ *BuildGraph(r)*.

Step 2. *InitSet*($\mathcal{G}$,E).

Step 3. for $i = 1$ to n do

1. $E_O := E$.
2. *EditDist1*($\mathcal{G}$,E,E_O,i),
3. *EditDist2*($\mathcal{G}$,E,E_O,i),
4. if $E[T_f] \leq d$, then return $(i, E[T_f])$.

図 4: The algorithm *APPROX*

Procedure InitSet($\mathcal{G}$,E)
$\mathcal{G} = (\mathcal{Q}, \mathcal{E})$: CMT transition graph;
E: array of edit distance;

1. $E[T_0] := 0$,
2. for all CMTs $T \in \mathcal{Q}$ in topological order do
   if T is an ϵ-node, then $E[T] := E[\overline{Pre}(T)]$; otherwise $E[T] := E[\overline{Pre}(T)] + 1$,
3. $update := 1$,
4. while $update = 1$ do
   (a) $update := 0$,
   (b) for all CMTs $T \in \mathcal{Q}$ in topological order do
      i. if T is an ϵ-node, then $E[T] := E[Pre(T)]$; otherwise $E[T] := E[Pre(T)] + 1$,
      ii. if $E[T]$ has been updated then $update := 1$.

図 5: The Procedure *InitSet*

(c) for any CMT T such that $T \neq T_0$ and $1 \leq h \leq L$,
$E^h[T,i] = \min\{[E^h[\overline{Pre}(T), i], E^{h-1}[Pre(T), i]\} + \gamma$, where if T is an S-node, then $\gamma = 1$; otherwise 0,

(d) for any CMT T such that $T \neq T_0$, $E[T,i] = E^L[T,i]$.

The number L is the maximum number of edges belonging to $\mathcal{E}_c$ over all loop-free paths on $\mathcal{G}$. The main algorithm *APPROX*, given in Fig. 4, computes the case of $i = 0$ by *InitSet*, the cases of 2-(a) and 2-(b) by *EditDist1*, and the case of 2-(c) by *EditDist2*. It is clear that we need $E[T,i]$ and $E[T,i-1]$ to compute $E[T,i]$ for any $1 \leq i \leq n$. Hence it is sufficient to use only two arrays $E[T]$ and $E_O[T]$ for $E[T,i]$ and $E[T,i-1]$. Furthermore, $E[Pre(T)] = \min_{T' \in Pre(T)}\{E[T']\}$ and $E[\overline{Pre}(T)] = \min_{T' \in \overline{Pre}(T)}\{E[T']\}$. $E[Pre(T)]$ and $E[\overline{Pre}(T)]$ are also defined similarly. We have the following Lemma 2 and Theorem 1.

Lemma 2 *The above recurrence correctly computes* $E[T,i]$.

Theorem 1 *Given an extended regular expression* r *of length* m, *a text string* x *of length* n *and any number* $d \geq 0$, *the algorithm APPROX correctly find all the prefix* y *of* x *with* $edit(y,r) \leq d$ *in* $O(N_r^2(2m)^{2N_r} + LN_r^2(2m)^{N_r}n)$ *time and* $O(N_r^2(2m)^{N_r})$ *space.*

Since L is less than the number of CMTs in $\mathcal{G}$, the algorithm APPROX runs in $O(N_r^2(2m)^{2N_r}n)$ time and $O(N_r^2(2m)^{N_r})$ space in the worst case. Thus if N_r is a constant number, then *APPROX* runs in polynomial time. In fact, if the number of extended operators occurring in r is a constant, then it runs in polynomial time because $N_r = 2u(r) + n(r) + 1$. Furthermore, it runs in $O(m^2 + mn)$ time and $O(m)$ space for regular expressions because $\mathcal{G}$ becomes just the NFA transformed from a regular expression and $N_r = 1$ and $L = 1$.

Procedure EditDist1($\mathcal{G}$,E,E_O, i)
$\mathcal{G} = (\mathcal{Q}, \mathcal{E})$: CMT transition graph; E, E_O: arrays of edit distance;

1. $E[T_0] := i$,
2. for all CMTs $T \in \mathcal{Q}$ in topological order do
   (a) if T is an ϵ-node, then $E[T] := E[\overline{Pre}(T)]$,
   (b) if T is an S-node with a symbol a, then $E[T] := \min\{E_O[T]+1, E_O[Pre(T)]+\gamma, E[\overline{Pre}(T)]+1\}$, where if a_i matches a then $\gamma = 0$; otherwise 1.

図 6: The Procedure *EditDist1*

Procedure EditDist2($\mathcal{G}$,E,E_0, i)
$\mathcal{G} = (\mathcal{Q}, \mathcal{E})$: CMT transition graph; E, E_O: arrays of edit distance;

1. $update := 1$,
2. while $update = 1$ do
   (a) $update := 0$,
   (b) for all CMTs $T \in \mathcal{Q}$ in topological order do
      i. if T is an ϵ-node, then $E[T] := E[Pre(T)]$; otherwise $E[T] := E[Pre(T)]+1$,
      ii. if $E[T]$ has been updated then $update := 1$.

図 7: The Procedure *EditDist2*

参考文献

[1] A.V. Aho, Algorithms for finding patterns in strings, In J.V. Leeuwen, ed. Handbook of theoretical computer science, Elsevier Science Pub., 1990.

[2] A. Apostolico, Z. Galil ed., Pattern Matching Algorithms, Oxford University Press, 1997.

[3] R.A. Baeza-Yates and G.H. Gonnet, Fast Text Searching for Regular Expressions or Automaton Searching on Tries, J. ACM, 43,6, 915-936, 1996.

[4] J.R. Knight and E.W. Myers, Super-Pattern matching, Algorithmica, 13, 1-2, 211-243, 1995.

[5] J.E. Hopcroft and J.D. Ullman, Introduction to automata theory language and computation, Addison Wesley, Reading Mass, 1979.

[6] G. Myers, A Four Russians Algorithm for Regular Expression Pattern Matching, J. ACM, 39,4, 430-448, 1992.

[7] E. Myers and W. Miller, Approximate Matching of Regular Expressions, Bull. of Mathematical Biology, 51, 1, 5-37, 1989.

[8] S. Wu and U. Manber, Fast Text Searching Allowing Errors, Communications of the ACM, 35, 10, 83-91, 1992.

[9] S. Wu, U. Manber and E. Myers, A Sub-Quadratic Algorithm for Approximate Regular Expression Matching, J. of Algorithm, 19, 346-360, 1995.

[10] H. Yamamoto, A New Recognition Algorithm for Extended Regular Expressions, ISAAC2001 Proc., LNCS 2223, 257-267, 2001.