JOURNAL OF ALGORITHMS 6, 132–137 (1985)

Finding Approximate Patterns in Strings

ESKO UKKONEN

Department of Computer Science, University of Helsinki, Tukholmankatu 2, SF-00250 Helsinki 25, Finland

Received October 14, 1983

Let p (the pattern) be a string and $t \geq 0$ an integer. The problem of locating in any string a substring whose edit distance from p is at most a given constant t is considered. An algorithm is presented to construct a deterministic finite-state automaton that solves the problem. © 1985 Academic Press, Inc.

1. INTRODUCTION

A classical *pattern matching problem* is, given strings p and x, to determine whether the *text* x contains an occurrence of the *pattern* p as a substring, that is, whether x can be written as $x = ypy'$. This is a well-studied question which can be solved, for example, by the Knuth–Morris–Pratt algorithm or by the Boyer–Moore algorithm [1, 2]. Denoting with m the length of p and with n the length of x, the Knuth–Morris–Pratt algorithm runs in $O(m + n)$ steps from which $O(m)$ steps are needed for preprocessing p and $O(n)$ steps are needed for the final scanning of x. On the average, the Boyer–Moore algorithm is even faster.

In this paper we are interested in an *approximate pattern matching problem* whose instance is specified by giving strings p and x and an integer t. Now one wishes to know whether x contains a substring p' that resembles p in the sense that the *edit distance* from p to p' is at most t. To define the edit distance, let the editing operations be insertion, deletion, and change, defined as follows: Inserting a symbol b at position i of a string $a_1a_2 \cdots a_m$ gives $a_1 \cdots a_ibA_{i+1} \cdots a_m$, deleting the symbol a_i at position i gives $a_1 \cdots a_{i-1}a_{i+1} \cdots a_m$ and changing the symbol a_i to another symbol b gives $a_1 \cdots a_{i-1}ba_{i+1} \cdots a_m$. The edit distance from a string u to a string v is defined as the minimum total number of such editing steps needed to convert u into v, cf. [1, 4, 5].

0196-6774/85 $3.00

Sellers [4] describes an algorithm which can be used to solve the approximate pattern matching problem. The algorithm is as follows. Suppose that $p = a_1 \cdots a_m$ and $x = b_1 \cdots b_n$. Evaluate an $(m+1) \times (n+1)$ matrix (d_{ij}), $0 \leqslant i \leqslant m$, $0 \leqslant j \leqslant n$, defined by the recursion

$$\begin{aligned} d_{0j} &= 0, \qquad 0 \leqslant j \leqslant n, \\ d_{i0} &= i, \qquad 0 \leqslant i \leqslant m, \\ d_{ij} &= \min(\textbf{if } a_i = b_j \textbf{ then } d_{i-1,j-1} \textbf{ else } d_{i-1,j-1} + 1, \\ &\qquad d_{i-1,j} + 1, d_{i,j-1} + 1), \qquad \text{otherwise.} \end{aligned} \tag{1}$$

Clearly, (d_{ij}) can be evaluated row-by-row or column-by-column, starting from the first row or from the first column, given directly by (1). After evaluating (d_{ij}) the algorithm looks at the values on the last row of (d_{ij}). From the results of Sellers [4, Theorem 1] it follows that some value, say d_{mj}, on the last row is at most t if and only if string x contains a substring p', ending at b_j, whose edit distance from $p = a_1 \cdots a_m$ is at most t. In fact, d_{mj} equals the edit distance between p and this substring of x. To locate p', note first that the dependencies between entries d_{ij} can be illustrated by drawing a directed arc from $d_{i'j'}$ to d_{ij} if and only if the minimization step in (1) gives d_{ij} from $d_{i'j'}$. Moreover, there is directed arc from $a_{i-1,0}$ to a_{i0} for all $i = 1, \ldots, m$. The resulting directed graph is called the *dependency graph*. Now, substring p' can be located by finding an element d_{0k} on the first row such that the dependency graph contains a directed path from d_{0k} to d_{mj}. Then we can choose $p' = b_{k+1} \cdots b_j$.

By evaluating (d_{ij}) and checking its last row for entries that are at most t, we can solve the approximate pattern matching problem. Obviously, the algorithm needs time $O(mn)$. The purpose of this paper is to further develop this algorithm such that the computations are divided into two phases: preprocessing the pattern p and scanning the text x. Although not too efficient, the preprocessing can be useful in some applications since after preprocessing, the scanning phase runs in time $O(n)$.

2. Preprocessing Algorithm

Given the pattern $p = a_1 \cdots a_m$ and the largest allowed edit distance t, the preprocessing algorithm constructs a deterministic finite-state automaton $M_p = (Q, \Sigma, h, q_0, F)$ that scans any text $x = b_1 \cdots b_n$ in Σ^* and arrives at some final state in F if and only if x contains a substring whose edit distance from p is at most t. Intuitively, each state of the automaton corresponds to a possible column which may occur in matrix (d_{ij}), defined

by (1), when the text x varies. All such columns as well as all transitions between adjacent columns for different symbols b_j of the text can be precomputed from p.

To give the precomputation algorithm, we denote with h the transition function of the finite automaton M_p. We use $S = (S_0, \ldots, S_m)$ and $S' = (S'_0, \ldots, S'_m)$ for arbitrary states (i.e., columns of (d_{ij})) of the automaton. The states are indexed by integers $0, 1, \ldots$. The index of state S is denoted as INDEX(S). In particular, the initial state q_0 is $(0, 1, \ldots, m)$ and INDEX$(0, 1, \ldots, m) = 0$. The alphabet of strings p and x is denoted by Σ. Besides h, the algorithm also computes the set F of the final states.

```
(1)  S := (0, 1, ..., m);
(2)  INDEX(S) := 0; i := 0;
(3)  NEW := {S}; STATES := {S};
(4)  F := if t ≥ m then {0} else ∅;
(5)  while NEW ≠ ∅ do
(6)      S := some element of NEW;                                  (2)
(7)      NEW := NEW − {S};
(8)      for each b ∈ Σ do
(9)          S' := NEXT-COLUMN(S, b);
(10)         if S' ∉ STATES then
(11)             NEW := NEW ∪ {S'};
                 STATES := STATES ∪ {S';
(12)             i := i + 1; INDEX(S') := i;
(13)             if S'_m ≤ t then F := F ∪ {INDEX(S')} endif;
                 endif;
(14)         h(INDEX(S), b) := INDEX(S');
         endfor;
     endwhile.
```

The procedure next-column, called on line (9), is

```
procedure NEXT-COLUMN(S, b):
(1)  S'_0 := 0;
(2)  for i := 1, ..., m do
(3)      S'_i := min(if a_i = b then S_{i-1} else S_{i-1} + 1, S'_{i-1} + 1, S_i + 1);
     endfor;
(4)  return (S'_0, ..., S'_m).
```

Algorithm (2) constructs finite-state automaton $M_p = (\{0, \ldots, k\}, \Sigma, h, 0, F)$, where k denotes the final value of i in (2).

Suppose that states $S = (S_0, \ldots, S_m)$ and $S' = (S_0', \ldots, S_m')$ are such that whenever $S_i \neq S_i'$ then $S_i, S_i' > t$. Then S and S' are equivalent states of our finite-state automaton, that is, the languages accepted from S and S' are identical. This is because the values d_{ij} on any directed path of the dependency graph of (d_{ij}) form a nondecreasing sequence. Hence if d_{ij} is $> t$, it cannot belong to a path leading to an entry on the last row of (d_{ij}) which is $\leqslant t$. This means that the exact values of the entries of a column S of (d_{ij}) (i.e., a state of our automaton) which are $> t$, have no influence on whether or not the last entry of the columns reachable from s is $\leqslant t$. In other words, S and S' are equivalent.

This observation reduces the number of the states of the automaton and can be implemented by adding the following sentence into procedure NEXT-COLUMN after line (3).

(3½) **if** $S_i' > t + 1$ then $S_i' := t + 1$;

In finding a natural data structure for the states of M_p in algorithm (2) as well as in analyzing the complexity of the algorithm, the following lemma is useful.

LEMMA 1. *Let* $S = (S_0, S_1, \ldots, S_m)$ *be a state of* M_p. *Then* $S_i - S_{i-1}$ *equals* -1, 0, *or* 1, *for* $i = 1, \ldots, m$.

Proof. Stated another way, Lemma 1 says that in matrix (d_{ij}), $d_{ij} - d_{i-1,j}$ always equals -1, 0, or 1. From (1) it immediately follows that $d_{ij} - d_{i-1,j} \leqslant 1$. Since all values d_{ij} are integers, the proof is complete if we show that $d_{ij} - d_{i-1,j} \geqslant -1$. This is proved by induction on the column index j. For $j = 0$ the claim is immediately true by (1). For a general $j > 0$, assume that $d_{ij} - d_{i-1,j} \leqslant -2$. Since $d_{i-1,j-1}$ is $\geqslant d_{i-1,j} - 1$ by (1), and since $d_{i,j-1}$ is $\geqslant d_{i-1,j-1} - 1$ by the induction hypothesis, we get from (1), that $d_{ij} \geqslant \min(d_{i-1,j-1}, d_{i-1,j} + 1, d_{i,j-1} + 1) \geqslant \min(d_{i-1,j} - 1, d_{i-1,j} + 1, d_{i-1,j} - 1) = d_{i-1,j} - 1$. Hence $d_{ij} - d_{i-1,j} \geqslant -1$. We have a contradiction which completes the proof. □

As regards detailed implementation of algorithm (2), we comment on lines (7), (10), and (11). Line (10) of the algorithm tests whether a generated state S' has been found earlier. The states found so far are represented as set STATES. Lemma 1 suggests a convenient implementation for STATES as a ternary tree; also any other balanced search tree could be used, of course. In such a tree, there are at most three edges leaving each internal node. One of the edges is labeled with -1, one with 0, and one with $+1$. The address of a node in the tree is the concatenation of the labels of edges on the path from the root to the node. State $S = (S_0, \ldots, S_m)$ of M_p can be represented by the leaf of the tree whose address is $(S_1 - S_0, S_2 - S_1, \ldots, S_m - S_{m-1})$. The method also works when all values S_i larger than $t + 1$ are replaced by $t + 1$ (step (3½) of procedure NEXT-COLUMN).

So all elements of STATES can be represented by the leaves of a ternary tree of height n. The tree requires space $O(mk)$, where k denotes the total number of states, and each membership test and each insertion can be performed in $O(m)$ steps. Moreover, set NEW is easily implemented as a queue of pointers to appropriate leaves of the tree for STATES. The queue needs space $O(k)$, and each insertion and deletion on lines (7) and (11) needs a constant time. Noting in addition, that each call of NEXT-COLUMN takes time $O(m)$, a straightforward calculation shows that the running time of algorithm (2) is $O(m \cdot |\Sigma| \cdot k)$. Moreover function h can be represented as an array of size $O(|\Sigma| \cdot k)$. The total space complexity is therefore $O((|\Sigma| + m) \cdot k)$.

To finally estimate k, the ternary tree representation of STATES immediately implies that $k = O(3^m)$. This is, however, a rough bound which omits the dependency of k on t and $|\Sigma|$. To derive an improved bound, recall that k actually denotes the number of different columns of matrix (d_{ij}) when p is fixed but x may vary, and all entries d_{ij} that would be $> t + 1$ are represented as $t + 1$. We partition all such columns $S = (S_0, \ldots, S_m)$ into disjoint classes. Class m contains all columns S such that $S_m \leqslant t$, and class i, $i < m$, contains all columns S such that $S_i = t$ and $S_j = t + 1$ for $j > i$. As already mentioned, if matrix (d_{ij}) contains a column S in class m (or equivalently, M_p enters a state in class m) then the text x contains a substring p', ending at this column, whose edit distance from p is at most t. It is easy to show that p' uniquely determines S. Hence, by counting the number of different edit operation sequences of length at most t, that are applicable on p, we see that class m contains at most $(|\Sigma| + 1)^t \cdot (2n + 1)^t$ elements. Similarly, the size of class i is bounded by the number of different strings p' whose edit distance from $a_1 \cdots a_i$ is t. We get a bound $(|\Sigma| + 1)^t \cdot (2i + 1)^t$. This means that $k = O(2^t \cdot |\Sigma|^t \cdot m^{t+1})$. We have shown

THEOREM 1. *Given a pattern $p = a_1 \cdots a_m$ in alphabet Σ and a maximum edit distance t, the preprocessing algorithm* (2) *constructs an approximate pattern matching automaton M_p in time $O(m \cdot |\Sigma| \cdot K)$ and in space $O((|\Sigma| + m) \cdot K)$, where $K = \min(3^m, 2^t \cdot |\Sigma|^t \cdot m^{t+1})$. The size of M_p is $O(|\Sigma| \cdot K)$.*

3. PRACTICAL IMPROVEMENTS

When m and k are not quite small, the large time and space requirements may limit the applicability of algorithm (2). We can, however, further develop the idea of preprocessing, based on the next lemma; the proof is similar to that of Lemma 1.

LEMMA 2. *In matrix* (d_{ij}), $d_{ij} \geqslant d_{i-1,j-1}$.

Hence the values on each diagonal of (d_{ij}) form a nondecreasing sequence. This leads to the following improved procedure for evaluating (d_{ij}): Evaluate (d_{ij}) from (1), column-by-column. During evaluation, maintain a pointer to the last entry of the column that is $\leqslant t$. If the pointer has value i for the current column (the possible values of i are $0, 1, \ldots, m$), then in evaluating the next column, it suffices to explicitly compute only entries $0, \ldots, i+1$. The remaining entries are known to have value larger than t, that is, they have the default value $t+1$.

It should be quite obvious, that for random pattern and text, the expected value of the pointer is $O(t)$. Hence the procedure can be used to solve the approximate pattern matching problem in expected time $O(tn)$, instead of time $O(mn)$ of the straightforward method.

The $O(tn)$ method also suggests that it is wasteful to preprocess the transitions between entire columns of (d_{ij}). A pattern matching machine which is almost as fast as M_p can be constructed by preprocessing the transitions between, say, the first $\frac{3}{2} \cdot t$ entries of each column of (d_{ij}). The resulting machine is as M_p but it must also explicitly maintain the entries on the column that are $\leqslant t$ and do not belong to the preprocessed area; we omit the details. On the average, such extra entries do not occur, hence the machine scans the text x in expected time $O(n)$.

REFERENCES

1. R. S. BOYER AND J. S. MOORE, A fast string matching algorithm, *Comm. ACM* **20** (1977), 262–272.
2. D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, Fast pattern matching in strings, *SIAM J. Comput.* **6** (1977), 323–350.
3. V. I. LEVENSHTEIN, Binary codes capable of correcting deletions, insertions and reversals. *Soviet Phys. Dokl.* **10** (1966), 707–710.
4. P. H. SELLERS, The theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms* **1** (1980), 359–373.
5. R. WAGNER AND M. FISCHER, The string-to-string correction problem, *J. Assoc. Comput. Mach.* **21** (1974), 168–178.