IMPROVING THE RUNNING TIMES FOR SOME STRING-MATCHING PROBLEMS

Sun Wu
Udi Manber
Eugene Myers

TR 91-20

August 1991

Improving the Running Times for Some String-Matching Problems

Sun Wu, Udi Manber[1], and Eugene Myers[2]

Department of Computer Science
University of Arizona
Tucson, AZ 85721

August 1991

Keywords: algorithm, approximate string matching, finite automata, regular expressions, sequence comparisons.

ABSTRACT

We present new algorithms for three basic string-matching problems: 1) An algorithm for approximate string matching of a pattern of size m in a text of size n in worst-case time $O(n + nm/\log n)$, and average-case time $O(n + nd/\log n)$, where d is the number of allowed errors. 2) An algorithm to find the edit distance between two sequences of size n and m ($n > m$) in time $O(n + nd/\log n)$, where d is the edit distance. 3) An algorithm for approximate matching of a regular expression of size m in a text of size n in time $O(n + nm/\log_{d+2} n)$, where d is the number of allowed errors. The last algorithm is the first $o(mn)$ algorithm for approximate matching to regular expressions.

1. Introduction

String-matching problems are becoming increasingly important parts of many applications. They appear in diverse areas such as molecular biology, information systems, pattern recognition, and text processing, just to name a few. In this paper we address three basic string-matching problems that have been studied extensively. We present new algorithms that generally obtain an $O(\log n)$ speedup over previous algorithms. We assume a unit-cost RAM model, in which arithmetic operations on $O(n)$-size numbers and addressing in an $O(n)$-size memory can be done in constant time. This model, of course, holds in most practical situations.

Let $A = a_1 a_2 a_3 ... a_n$ and $B = b_1 b_2 b_3 ... b_m$, such that $n \geq m$, be two sequences of characters from a finite fixed alphabet Σ. An *edit script* from B to A is a sequence of insertions, deletions, and/or substitutions to B that result in A. The problem of determining a shortest edit script (SES) between two sequences of symbols has been studied extensively ([Hi75, HS77, My86, NKY82,

[1] Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, and by an NSF grant CCR-9002351.

[2] Supported in part by the NIH (grant R01 LM04960-01), and by an NSF grant CCR-9002351.

Uk85b, WF74, WMMM90] is a partial list). The classic dynamic programming algorithm by Wagner and Fischer [WF74] has $O(mn)$ worst-case running time. Masek and Paterson [MP80] were the first to improve the $O(mn)$ worst case by using the "Four-Russians" technique [ADKF70] to reduce the worst-case running time to $O(n + mn/\log n)$. This is the best known running time in terms of n and m. However, several algorithms have been designed that are much faster under some conditions. Hunt and Szymanski [HS77] presented an algorithm whose running time is $O(R \log\log n)$, where R is the total number of ordered pairs of positions at which the two sequences match (see also [AG87]). Myers [My86], Ukkonen [Uk85b], and Nakatsu, et al. [NKY82] gave algorithms with worst-case time complexity $O(nd)$, where d is the length of an SES, which are efficient when A and B are similar. Myers further showed how to refine his version of the result to take $O(n \log n + d^2)$ time in the worst case using suffix trees and $O(1)$ lca finding. Wu et. al. [WMMM90] improved the running time slightly to an $O(np)$ algorithm where p is the number of deletions in the shortest edit script (which could be much smaller than d when m is much smaller than n). We present here a new algorithm whose running time is $O(n + nd/\log n)$ in the worst case.

The approximate string-matching problem is a similar problem, with the difference being that B is to be matched *inside* A rather than to all of A. Also, we assume that the number of allowed errors (insertions, deletions, or substitutions), d, is given to us, and the goal is to find all positions in A that match B within d errors. The dynamic programming approach extends easily to solve the approximate string-matching problem with the same $O(mn)$ complexity [Se80]. Several algorithms achieve $O(nd)$ running time for the approximate string-matching problem [GG88, GP90, LV88, LV89, Uk85a], which is better than the dynamic-programming approach for small d, but not in the worst case. Masek and Paterson results [MP80] also apply to the approximate string-matching problem with the running time of $O(n + nm/\log n)$. (The log factor is actually $\log_c n$ for $c = O(|\Sigma|^2$.) Our algorithm achieves the same worst-case running time, $O(n + nm/\log n)$, but it is simpler, and it serves as a basis for the other algorithm. Its average-case running is $O(n + nd/\log n)$, which is better than Masek and Paterson's algorithm. Moreover, our algorithm is better in practice. Whereas Masek and Paterson showed that their algorithm improves the direct dynamic programming algorithm only when n is larger than 262,419, our approach of a universal, precompiled table delivers a log-factor speedup for *all* choices of n.

The approximate regular-expression matching problem is similar to the approximate string-matching problem, except that instead of a simple string as a pattern we are given a regular expression. We want to find all the substrings in A that are within d errors to strings that can be generated by the regular expression. Algorithms with running time of $O(mn)$ have been given by [WS78] and [MM89]. Wu and Manber [WM91] presented another algorithm (which is part of the *agrep* package) whose worst case running time is $O(nd + mnd/\log n)$, but which performs very fast in practice (where usually d and m are quite small). Myers [My88] has used the 4-Russians technique to speed up the *exact* regular expression matching problem (i.e., no errors are allowed) to $O(mn/\log n)$. Speeding up the approximate case is listed as the major open question in his paper. In this paper we present a new algorithm for the approximate matching of regular expressions with a running time of $O(n + nm / \log_{d+2} n)$.

2. The Main Approach

The problems we address have a common feature in that there is a *text* and a *pattern* and we find the matching by scanning the text, character by character, recording some information and looking for matches. Most string-matching algorithms operate that way. The question is what information to maintain and how to process it. We take a general approach. We model the scanning by an automaton. In each step we scan one character and model the information we have so far as a state in the automaton. Thus, processing the next step after seeing the next character corresponds to moving in the automaton from one state to another. The main problem, of course, is to find a good short encoding of the states and a good fast traversal algorithm for the automaton.

Let's assume that the number of states that the automaton, corresponding to a particular problem, has is $S(m)$. If $S(m)$ is not too large, we can precompute the whole transition table, which is a directed graph with the states as nodes, and then implement the automaton traversal by table lookup. But usually $S(m)$ is very large. For example, if we follow the dynamic-programming approach for the sequence comparison problem, in each step we maintain a column of size m with the sizes of the best matches of all prefixes of the pattern. Since matches can potentially cost as much as m, the number of states in the automaton is m^m. Not only are we prevented from storing the whole transition table, but each state requires $m \log m$ bits to represent so each table lookup would take (at least) $O(m)$. (And we can compute the next step in $O(m)$ time without having the whole table anyway.) Our approach is first to reduce the number of states as much as we can (e.g., we will show that for the edit-distance problem 3^m states are sufficient) and second to find a way to partition the set of states into regions such that 1) the transition table within each region (i.e., the subgraph) is small enough so that it can be precomputed and stored and 2) the transition between two (global) states can be computed by combining the transitions between regions (modified in some fashion) in an efficient way.

Furthermore, except for the regular expression case, we use a *universal* automaton such that every region uses the same automaton. Using a universal automaton is not crucial for the approximate string-matching problem (although it simplies the algorithm), but it is crucial for the sequence-comparison problem. If there are $S(m)$ states then each state is encoded by at least $\log_2 S(m)$ bits. We divide this encoding into blocks of size K (whose value depends on the problem). Given the current state and the next text character, we start with the first block and compute a bit vector that is used in the transition of the universal automaton to find the new block (which is the first block in the new state) and some output that is then used (along with the text character) to compute another bit vector with the second block, and so on. We use the special properties of the automata corresponding to our problems to achieve these goals (arbitrary graphs cannot be partitioned so nicely). We use different encodings and different partitions for the three problems we address, but the main idea is the same.

We start with the approximate string-matching problem, whose solution is the simplest among the three problems. We then use the same partition scheme, but a more efficient traversal procedure, for the sequence comparisons problem. We end with the most complicated algorithm, the one for regular expressions, which uses a whole different partition scheme.

3. Approximate String Matching

The input to the problem is a text $A = a_1 a_2 a_3 ... a_n$, a pattern $B = b_1 b_2 \cdots b_m$, and d the number of allowed errors. We want to find all the substrings in A that are within d errors to B. We start with the regular dynamic-programming algorithm. Let $A[j]$ denote the sequence $a_1 a_2 a_3 ... a_j$ (the j'th prefix of A) and let $B[i]$ denote the sequence $b_1 b_2 b_3 ... b_i$. We compute an $m \times n$ matrix E such that $E[i, j]$ is the cost of the smallest SES between $B[i]$ and a suffix of $A[j]$. The matrix is computed using the following recurrence [Se80]:

$$E[0, j] = 0 \quad \text{for } 0 \le j \le n; \quad E[i, 0] = i, \quad \text{for } 0 \le i \le m.$$

$$s_{ij} = \begin{cases} 0 & \text{if } b_i = a_j \\ 1 & \text{otherwise} \end{cases}$$

$$E[i, j] = \min\left(E[i-1, j-1] + s_{ij}, E[i-1, j] + 1, E[i, j-1] + 1\right). \tag{3.1}$$

Ukkonen [Uk85a] used an automaton such that each column vector is treated as a state, and the next column is obtained by a state transition of the automaton. He showed that the automaton has $\le 3^m$ states, which is implied by the next lemma.[3]

Lemma 3.1: [Uk85a]

$$-1 \le E[i, j] - E[i-1, j] \le 1 \quad \text{for } 1 \le i \le m,\ 0 \le j \le n. \tag{3.2}$$

□

We carry this idea a little further by using the differences $D[i, j] = E[i, j] - E[i-1, j]$ directly in the recurrence and maintaining only these values. (The c_i values used in the recurrence depend on j, but since they are computed separately for each column, we index them with the row number only for simplicity.)

Lemma 3.2:

$$c_0 = 0; \quad D[0, j] = 0, \text{ for } 0 \le j \le n; \quad D[i, 0] = 1, \text{ for } 1 \le i \le m;$$

[3] If substitutions are not allowed (e.g., we want the longest common subsequence), then we can reduce the number of states to 2^m, and make the algorithm simpler to implement.

$$D[i,j] = \begin{cases} 1 + \min\left[\min(c_{i-1}, c_{i-1} + D[i,j-1)]), 0\right] & \text{if } b_i \neq a_j \\ c_{i-1} & \text{if } b_i = a_j \end{cases} \quad (3.3)$$

$c_i = c_{i-1} + D[i,j-1] - D[i,j]$.

Proof: We have to prove that (3.3) is equivalent to (3.1). First note that $c_i = \sum_{k=1}^{i}\left[D[k,j-1] - D[k,j]\right]$. Thus, c_i is equal to $E[i,j-1] - E[i,j]$. The proof is by induction on j and i. Assume that the recurrence is correct up to $j-1$ for all i, and for j up to $i-1$, and consider $D[i,j]$. There are two cases.

$b_i \neq a_j$

There are two subcases. If $E[i-1,j] \le \min(E[i-1,j-1], E[i,j-1])$ (i.e., a deletion is in order), then if we use (3.1) we get $E[i,j] = E[i-1,j]+1$, which implies that $D[i,j]=1$. Now if we use (3.3) and the induction hypothesis, we get $c_{i-1} \ge 0$ and $c_{i-1} + D[i,j-1] \ge 0$, thus from (3.3) we get that $D[i,j] = 0 + 1 = 1$. Otherwise, let $\bar{E}$ denote the minimum of $E[i-1,j-1]$ and $E[i,j-1]$, which implies (by (3.1)) that $E[i,j] = \bar{E}+1$. We have $D[i,j] = E[i,j] - E[i-1,j] = \bar{E} + 1 - E[i-1,j] =$ (by definition of $\bar{E}$) $1 + \min((E[i-1,j-1] - E[i-1,j]), (E[i,j-1] - E[i-1,j])) = 1 + \min(c_{i-1}, c_{i-1} + D[i,j-1])$, which is exactly $D[i,j]$ as defined by (3.3).

$b_i = a_j$

In this case, since $E[i,j] = E[i-1,j-1]$, $D[i,j] =$ (by (3.1)) $E[i,j] - E[i-1,j] = E[i-1,j-1] - E[i-1,j] = c_{i-1}$, which is exactly (3.3). □

A global state of the automaton can be represented by a vector of size m whose entries are -1, 0, or 1. We divide the global states in a natural way. Each m-vector is divided into subvectors, called *regions*, of size r (to be determined later). Each region corresponds to a string of size r, thus it can have 3^r possible values. We call the values of a region its state (from now on we will use the term state to represent a region rather than a global state of the automaton). We encode the state of a region by an integer. We would like to compute the state transition for each region in constant time. In other words, we would like to use (3.3) instead of going from $i-1$ to i (for a given j), going from $i-1$ to $i+r-1$ (r values of the column vector) in one step. Let's study (3.3) more closely. The values of $D[i,j]$ to $D[i+r-1,j]$ depend on 1) the values of $D[i,j-1]$ to $D[i+r-1,j-1]$; 2) the value of c_{i-1}; and 3) the values of s_{ij} to $s_{(i+r-1)j}$, called characteristic vector. Overall, they depend on two vectors of size r, and the 'carry' c_{i-1}. Both the column vector and characteristic vector (of size r) can be encoded by an integer in an obvious way. We can build a function (corresponding to a universal automaton) which, given any such $2r+1$ values encoded as three integers, outputs the following values: c_{i+r-1}, an integer corresponding to $D[i,j]$ to $D[i+r-1,j]$, and $\sum_{k=i}^{i+r-1} D[k,j]$ (so that $E[n,j]$ can be computed). If $r = \frac{1}{2}\log_3 n$, then it is easy to see that the function has $O(3^{\frac{1}{2}\log_3 n} \cdot 2^{\frac{1}{2}\log_3 n}) = o(n)$ possible inputs and it can be computed in time $O(n)$ (we can actually make r slightly larger; in practice, the value of r should be determined by the amount of available space and n). We can precompute the transition table for this function and store it. Since this is a universal automaton, it needs to be computed only once for each n (or range of values for n).

Using the precomputed transition table, we can compute $r = O(\log n)$ values of the dynamic-programming table in constant time if the input to the transition table can be obtained in constant time. The problem is the characteristic function. We need r binary values for each input, and computing these values from scratch every time will take r steps. Instead, we utilize the fact that there can be at most $|\Sigma|$ different characters in the text and compute characteristic vectors for each $\sigma \in \Sigma$. In other words, we build a two-dimensional table of dimensions $|\Sigma| \times m/r$, such that for each $\sigma \in \Sigma$ we can obtain the m/r vectors of the characteristic function corresponding to $a_j = \sigma$ directly.

After the transition table and the characteristic function table have been built, the algorithm finds the approximate matches in the following way. Initially the state of each region p, $1 \le p \le \lceil m/r \rceil$, are set to the value corresponding to $(1, 1, 1, ..., 1)$. We scan the text one character at a time. For every character a_j being scanned, we compute the state transition for every region p in increasing order of p, and accumulate the column sum along the way. First, we find the characteristic value for a_j corresponding to region p by looking up the characteristic function table. Then, by using $c_{(p-1)r}$ (obtained from the computation in region $p-1$), the current state of region p, and the characteristic value, we look up the transition table to find the new state of region p, the next carry $c_{p \cdot r}$, and the column sum of the new state. The computation is continued, with the column sum being accumulated, until the last region. The algorithm reports a match whenever the accumulated column sum is $\le d$ (i.e., $E[m, j] \le d$). Since each region can be processed in constant time, and there are $O(m/\log n)$ regions in each column, the total time for scanning a character is $O(m/\log n)$. The total running time is thus $O(n + nm/\log n)$. The space needed is $O(n + m|\Sigma|/\log n)$.

The algorithm described above can be improved to be more practical. As was shown in [Uk85a], the expected time for the dynamic programming can be improved to $O(nd)$ by computing only a portion of the dynamic programming table. The approach is to compute, for each column j, a row f_j such that $E[f_j, j] \le d$ and $E[i, j] > d$ for all $i > f_j$. If we know the value of f_j, then we do not need to look at rows below it in column j. It is not hard to see that $-1 \le f_j - f_{j-1} \le 1$, and that f_j can be computed from f_{j-1}. (Notice that it is not sufficient to find the first E value that is $> d$, because following values can still be $\le d$.) Since we handle whole regions in each step, it is slightly more difficult to determine the last region that we need to compute. Let $s = (D[i, j], D[i+1, j], ..., D[i+r-1, j])$ be a state; the minimal prefix sum of s is $\min_p \sum_{i=1}^{p} D[i, j]$, $1 \le p \le r$. We keep the minimal prefix sum for each state, and add that information to the transition table. Let r_j denote the last region in column j that contains an entry whose E

value is $\leq d$. We assume that r_{j-1} is known; we compute r_j in the following way. Suppose we have computed regions up to region $r_{j-1}+1$ in column j. (If we need a state from the previous column that has not been computed, we use a default state equals to $(1,1,1,...,1)$.) There are three cases (all column sums and prefix sums refer to column j).

1. The column sum up to region r_{j-1} plus the minimal prefix sum of region $r_{j-1}+1$ is $\leq d$. In this case, we set $r_j = r_{j-1}+1$.
2. The condition of case 1 is not satisfied, and the column sum up to region $r_{j-1}-1$ plus the minimal prefix sum of region r_{j-1} is $> d$. In this case, we set $r_j = r_{j-1}-1$.
3. Otherwise, $r_j = r_{j-1}$.

These rules are easy to verify and we leave it to the reader. Since our approach has an $O(\log n)$ speedup over the dynamic programming approach, the expected time complexity of the improved algorithm is $O(n + nd/\log n)$.

4. Sequence Comparison

The sequence-comparison problem can be solved similarly to the approximate string-matching problem. The only differences from Lemma 3.2 is the boundary conditions which are

$$c_0 = -1; \quad D[0,j] = j, \text{ for } 0 \leq j \leq n, \tag{4.1}$$

and that we need only the value of $E[m,n]$. The computation can proceed in the same manner leading to $O(n + nm/\log n)$ running time. But we can do better when the two sequences are similar. For simplicity, we assume that $n = m$, so the matrix is square. Let d be the length of the SES between A and B. The standard dynamic programming approach computes all $D[i,j]$'s. However, if the SES is of size d then no match that occurs within distance of more than $d/2$ from the main diagonal will be used, because it will cost more than d insertions, deletions, or substitutions to get there and back to the main diagonal (which is where we have to end). Therefore, a band of width $d/2$ around the main diagonal contains all the information needed to compute $E[n,n]$. We would like to compute the values of the matrix only in that band. This is done by using the same partition of the set of states and the same recurrence (with a different boundary conditions) but with extra care as to how we scan the text.

Since the values beyond the d band are not used, we might as well assume that there are no character matches there. We will actually use a 'jagged' band of size at most $d+2r$ (where r is the size of a block) as is shown in Figure 1. Of course, the value of d is usually unknown, but this is not a major problem because we can use doubling.

The only part in this computation that is different from before is when we 'jump' from a square to one below (as in going from X to Y in Figure 1). Let i be the row at the beginning of square Y. The only values we need from the rows above is the column sum and c_{i-1}. The column sum is incremented by 1 from the previous column (at the same row position). And, since we can assume that there are no character matches above i at that column (they will not count anyway),

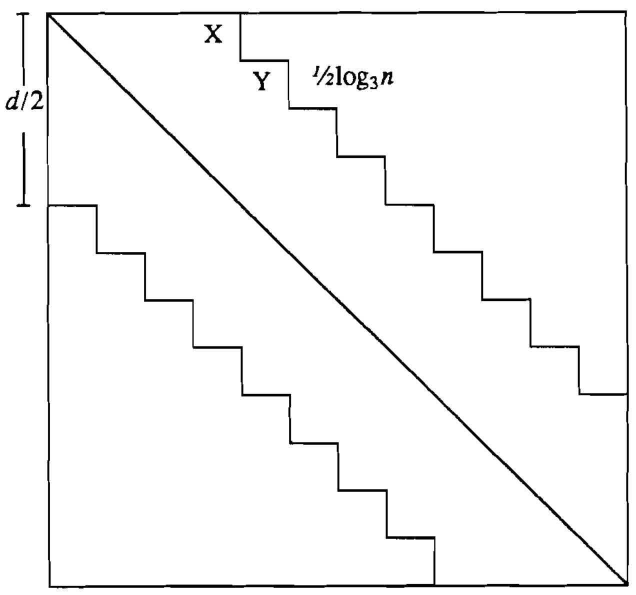

Figure 1: Working around the diagonal.

we assign $c_{i-1} = -1$, and continue accordingly. The edit distance we get is guaranteed to be correct if it is $\leq d$. Since we process at most $O(d)$ rows for each column, the running time is $O(n + nd/\log n)$ in the worst case (the preprocessing is the same as before).

5. Approximate Matching of Regular Expressions

The input to the problem now is a text $A = a_1 a_2 a_3 ... a_n$, a regular expression R of size m, and d the number of allowed errors. We want to find all the substrings in A that are within d errors to strings that can be generated by regular expression R. We use the same approach of dividing the pattern into parts and processing each part in constant time, but the partition (and the corresponding recurrence) is much more complicated. In a nutshell, we first use Thompson's construction of a non-deterministic finite automaton (NFA) for the regular expression [Th68], then partition the NFA into modules (following Myers' construction [My88]) such that the modules 'communicate' among themselves in a particular way. Each module is solved by two DFA's in an amortized fashion. We design the recurrences to take advantage of the construction. We then put it all together to improve the running time.

Let R be a regular expression, and let M be the corresponding NFA constructed using Thompson's construction. We call the nodes corresponding to characters L-nodes and nodes corresponding to ε-moves ε-nodes. We number the nodes by a topological order disregarding the back edges (which were formed by the closure operation). The *cost* of going from one node v to another node u in the NFA is the minimal number of L-nodes along a path from v to u (excluding

v). Let r_i denote the character corresponding to node i, and let $Pre(i)$ denote the nodes in M that are predecessors of node i. Let $\overline{Pre}(i) \subseteq Pre(i)$ denote the predecessors of i excluding back edges. Let $Pre^*(i)$ denote the closure predecessors of i,which is the set of L-nodes such that the shortest path from any node in $Pre^*(i)$ to i has cost 1 if i is an L-node, and cost 0 if i is an ε-node. $E[i, j]$ will again denote the cost of the edit distance from the a sub-pattern of R to a substring of A ending at a_j, except that now the sub-pattern corresponds to every string that can reach the ith node of the NFA from the start state. We compute $E[i, j]$ in a similar way to (3.1), with two exceptions. 1) Instead of $i-1$ in (3.1), we use $Pre(i)$ and/or $\overline{Pre}(i)$; 2) To avoid cycles in the recurrence (any NFA corresponding to a regular expression with a closure operation * will contain cycles), and to handle the ε moves, we use two passes. The value of $E[i, j]$ after the first pass is denoted by $E'[i, j]$. We denote by $E[Pre(i),j] = \min_{k \in Pre(i)} E[k,j]$; that is, the minimum edit distance to any predecessor of i; $E'[Pre(i),j]$ is defined similarly with E' replacing E, $E[\overline{Pre}(i),j]$ is defined similarly with $\overline{Pre}$ replacing Pre, and $E[Pre^*(i), j]$ is defined similarly with Pre^* replacing Pre. The exact recurrence is given next.

$$E[0,j] = 0 \ \text{ for } 0 \le j \le n; \quad E[i,0] = S(i), \ \text{ for } 1 \le i \le m,$$

where $S(i)$ is the shortest path from the start state of M to node i (moving to an ε node costs 0).

$$s_{ij} = \begin{cases} 0 & \text{if } r_i = a_j \\ 1 & \text{otherwise} \end{cases}$$

$$1.\ E'[i,j] = \begin{cases} \min\left[E[i,j-1]+1,\ E[Pre(i),j-1]+s_{ij},\ E'[\overline{Pre}(i),j]+1\right] & \text{if } i \ge 1 \text{ is an L-node} \\ E'[\overline{Pre}(i),j] & \text{if } i \ge 1 \text{ is an } \varepsilon\text{-node} \end{cases}$$

$$2.\ E[i,j] = \begin{cases} \min(E'[i,j],\ E[\overline{Pre}(i),j]+1) & \text{if } i \ge 1 \text{ is an L-node} \\ \min(E'[Pre(i),j],\ E[\overline{Pre}(i),j]) & \text{if } i \ge 1 \text{ is an } \varepsilon\text{-node} \end{cases} \qquad (5.1)$$

This recurrence is equivalent to Figure 6 in [MM89], and its proof follows from the discussion in [MM89]. We only outline the intuition behind it. The first pass for L-nodes handles insertions, substitutions/matches, and deletions (in that order), but only for edges in the forward direction (which is always the case for L-nodes). We cannot handle back edges in one pass, because they might come from nodes with higher labels, which we have not processed yet. The first pass for ε-nodes propagates the values obtain so far through ε-moves. Again, no back edges are used. After the first pass, the values of $E'[i, j]$ are equal to the desired $E[i, j]$, except for a possibility of a series of deletions on a path that includes back edges. The second pass handles such paths. If i is an ε-node, then in pass 2 it receives the best E' value from its predecessors including those connected by back edges. Thus, deletions will be propagated through at least one back edge. If i is an L-node, then we use the regular forward propagation for deletions. So, a series of deletions on a path with no more than one back edge will be handled. It turns out that one never has to use more than one back edge in such a propagation (see [MM89]). Figure 2 shows an example of computing $E[i, j]$ by using recurrence 5.1.

Recurrence (5.1) leads to an algorithm whose running time is $O(nm)$ in the worst case, because Thompson's construction guarantees that $|Pre(i)| \leq 2$ for all i. We can improve the running time by using our general technique. First, we reduce the number of states in the global deterministic finite automata from m^m to $(d+2)^m$ by allowing only $d+2$ values for the $E[i,j]$'s. If the edit distance is d, then there is no need to maintain values $> d$, and they can be replaced by $d+1$. We then decompose the NFA into modules, each of size $O(\log_{d+2} n)$, such that, when combined together, they can be used to simulate the behavior of the original algorithm. The technique is similar to the one we used in the previous two sections, but the implementation is much more complicated. We do not use a universal automaton here, because we will need to encode the structure of the module. (This is probably possible, but too complicated.)

To improve the algorithm we have to answer the following two questions: 1) how to decompose the NFA into appropriate modules, and 2) how to combine the modules to simulate the function of the original algorithm. The 'decomposition' part is about the same as in [My88], and we will describe it only briefly here. The 'combine' part is quite elaborate and will be described in detail.

The decomposition of the NFA for R takes advantage of the hierarchical form of regular expressions. For this reason, we will first express the decomposition in terms of the associated parse tree, T_R, for R. Hereafter, T refers to T_R whenever R can be inferred from context. We first partition T into a collection of subtrees. Then, we connect the subtrees in the following way. Let T_p and T_q be subtrees, and assume that T_p is connected to T_q by an edge (v, u) such that $v \in T_p$ and $u \in T_q$. We add a 'pseudo-node' to T_p to represent u. This 'pseudo-node' will serve to

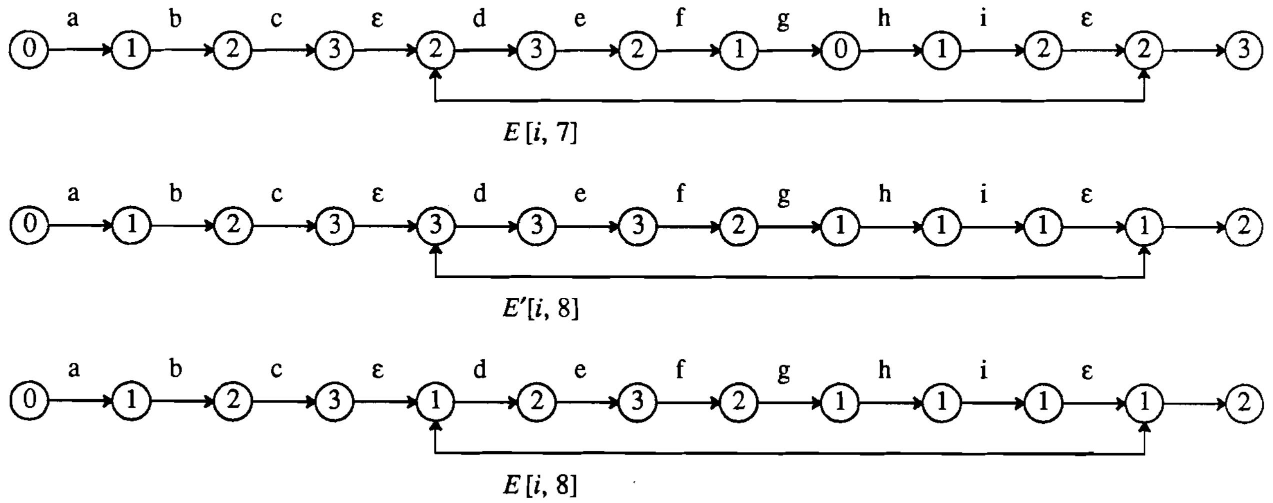

Figure 2: An example of computing $E[i, j]$; $R = \text{abc(defghi)}^*\text{j}$; $A = \text{abcdefgi...}$

communicate values between the subtrees. We call the subtrees (with the extra pseudo-nodes) *modules*. We will use the term 'original nodes' to indicate nodes that are in T (i.e., nodes that are not pseudo-nodes). Figure 3 shows an example of this decomposition. A square denotes a pseudo-node. The following lemma shows that the decomposition of T can be done evenly.

Lemma 5.1: For any $K \geq 2$, we can decompose T as described above into a module U that contains T's root and has no more than K nodes, and a set of other modules, denoted by X, each having between $\lceil K/2 \rceil$ and K nodes.

Proof: The proof is by induction on size of the tree. Suppose that the hypothesis is true for trees of size $m-1$, and consider a tree of size m. (The base case is trivial.) Let r be the root of T. There are two cases:

r has two children

Let c and d be the children of r and T_c and T_d be the subtrees rooted at c and d respectively. By the induction hypothesis, T_c and T_d can be decomposed into $U_c \cup X_c$ and $U_d \cup X_d$ respectively. Let k_c be the number of nodes in U_c, and k_d be the number of nodes in U_d. If $k_c + k_d < K$ then we can set $U = U_c \cup U_d \cup \{ r \}$, and $X = X_c \cup X_d$. Otherwise, without loss of generality, assume that $k_c \geq k_d$. This implies that $k_c \geq \lceil K/2 \rceil$. If $k_d < K$ then we can let $U = U_d \cup \{ r \}$, and $X = X_c \cup X_d \cup U_c$. Otherwise it must be that $k_c = k_d = K$, and we can let $X = X_c \cup X_d \cup U_c \cup U_d$ and $U = \{ r \}$.

r has one child

Let c be the only child of r, and T_c be the subtree rooted at c. By induction hypothesis, T_c can be decomposed into U_c and X_c. Let k_c be the number of nodes in U_c. If $k_c < K$ then we just let $U = U_c \cup \{ r \}$, and $X = X_c$. Otherwise, let $U = \{ r \}$ and $X = X_c \cup U_c$, and the proof

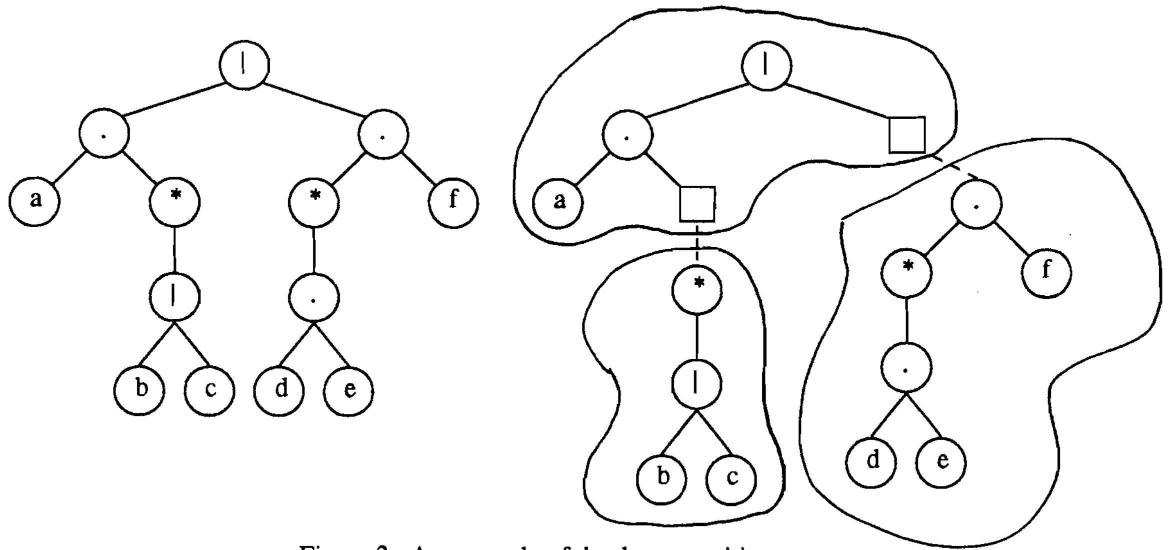

Figure 3: An example of the decomposition.

is completed. □

We can decompose the NFA for R in a way corresponding to the decomposition of R's parse tree such that a module in the NFA for R corresponds to a module in R's parse tree. Inside a module M_i, we will call a node an *original* node if that node is the original node in the NFA. Otherwise that node corresponds to a module inside M_i and is called a *pseudo-node*. Module M_i is called the *parent* module of M_j (and M_j is called the *child* module of M_i) if M_i contains the pseudo-node corresponding to M_j. A module that contains only original nodes is called a *leaf* module. A module that contains pseudo-nodes is called an *internal* module. It is not hard to see, based on Lemma 5.1, that given a constant K, we can decompose the NFA into a collection of modules such that 1) each module contains $\leq K$ nodes, (including original nodes and pseudo-nodes), 2) each module contains one *input* node and one *output* node which are used to communicate with its parent module, and 3) the total number of modules is bounded by $O(m/K)$. Figure 4 shows an example of decomposition for regular expression $R = a(b|c)^*|(de)^*f$. (All moves, except for the back edges, are from left to right); The NFA for R is decomposed into 3 modules, assuming each module can have up to 6 nodes.

In our algorithm, we choose $K = \frac{1}{2}\log_{d+2} n$. For each module obtained from the decomposition we build two transition tables corresponding to pass 1 and 2 of recurrence 5.1. Suppose we are to build the transition tables for M_i, which contains $t \leq K$ nodes. Suppose that g out of the t nodes in M_i are pseudo-nodes ($g=0$ if M_i is a leaf module). We label the nodes in a topological order ignoring the back edges. Node 0 is the input node, and node $t-1$ is the output node of M_i. We call the value of node i (whether it is after or before pass 1, that is, whether it corresponds to $E[i,j]$ or to $E'[i,j]$) the E value of i. A *state* s of M_i is a vector $(e_0, e_1, \ldots, e_{t-1})$, where e_i

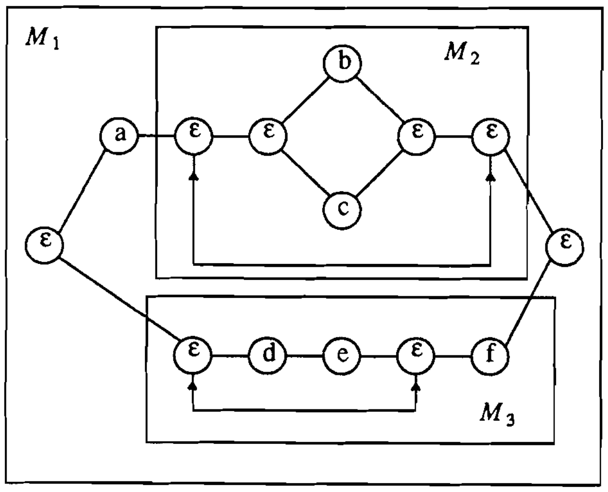

Figure 4: The NFA for $a(b|c)^*|(de)^*f$ and its decomposition.

denotes the E value of node i. We encode the state s by an integer I_s. In the encoding, I_s contains t components each containing $\lceil \log_2(d+2) \rceil$ bits. A component of I_s corresponds to an e_i, $0 \le i \le t-1$, in s. Hereafter, we will refer to e_i as the component in I_s that corresponds to e_i. Assume that x is a pseudo-node in M_i. Let M_x be the module corresponding to x. Suppose that u and v are original nodes in M_i, and that u is a predecessor of x and v is a successor of x. Note that the computation of the E value for the input node of M_x depends on the E value of u, and the computation of the E value for v depends on the E value of the output node of M_x. In other words, the computation of the E value for v has to wait until the computation in module M_x has been completed. The dependency on pseudo-nodes prohibits us from building a transition table that can be used to compute the state transition for M_i in constant time as in previous sections. We divide the state transition into a series of smaller transitions as follows. We partition M_i into h layers such that 1) nodes in layer p have labels greater than nodes in layer $p-1$; 2) no two pseudo-nodes that are connected by a path are in the same layer; 3) if x is a pseudo-node in layer p then its successor node is in layer $p+1$; 4) each layer either contains at least one pseudo-node or it contains the output node of M_i. (If M_i is a leaf module, then it has only one layer.) Figure 5 shows an example of partitioning a module into layers. Nodes represented in boxes are pseudo-nodes. The E values of nodes in layer p do not depend on pseudo-nodes inside p. So, we can build transition tables such that the computation of the E values in a layer can be done in constant time.

The state transition for M_i is decomposed into h serial steps, each corresponding to a state transition for one layer. In the preprocessing, for every possible state s (represented by I_s), every layer number p, $1 \le p \le h$, and every possible input character a_j, we precompute the state transition for layer p using pass 1 of recurrence 5.1 on the nodes in layer p. The result is stored in the first transition table $Next_1$. The building of the second transition table $Next_2$ is in the same way except that now the input character is not needed and pass 2 of recurrence 5.1 is now applied; namely, the second transition table $Next_2$ is addressed by current state and layer number only.

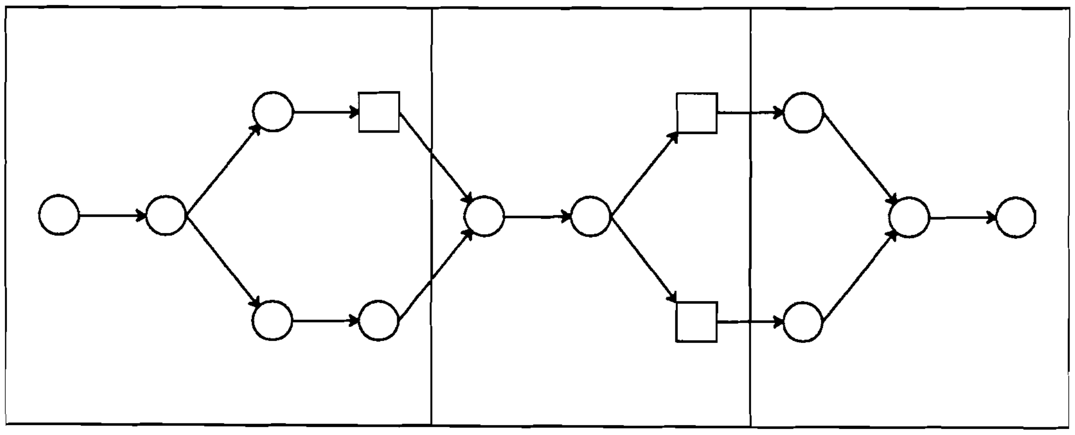

Figure 5: An example of the partitioning

With the transition tables described above, the state transition for M_i is done in two passes in the following way. We call the procedure for pass 1, *Transition*$_1$, and the procedure for pass 2, *Transition*$_2$. In *Transition*$_1$, we first apply a state transition for layer one of M_i using table *Next*$_1$ based on the current state, the input character, and the layer number, 1. Then we apply the state transitions for all child modules of M_i that are in layer one, by recursively calling *Transition*$_1$. The pseudo-nodes are used to communicate values between parent/child modules; we discuss this in the next paragraph. After that we apply the state transition for layer two of module M_i, based on the new state, the input character, and the layer number, 2. We continue in this way until every layer of M_i has been processed. After pass 1 is done, we apply *Transition*$_2$ in the same way except that the input character is not needed and *Next*$_2$ instead of *Next*$_1$ is used.

Before we use *Next*$_1$ for each layer of M_i, we have to ensure that the E values of the pseudo-nodes are consistent in the parent and child modules. Recall that a pseudo-node x represents a regular node in M_x, whereas in M_i x represents the module M_x. In M_x we have two special nodes, an input node and an output node. The E value of the input node of M_x should be the same as the E value of x in M_i before we apply the transition on the appropriate layer in M_i containing x. After we are done with the layer (which means that we have applied the transition to all its child modules), we copy of the E value of the output node of M_x to x. This way the E values are kept consistent between modules. The complete algorithm is is given in Figure 6. The algorithm reports a match whenever the E value of the output node of the outer most module is $\leq d$.

Analysis: Let's start with space complexity. In the decomposition we have chosen K, the maximum size of a module, to be $\frac{1}{2}\log_{d+2}n$. So, the maximum number of possible states for a module is $(d+2)^{\frac{1}{2}\log_{d+2}n} = n^{\frac{1}{2}}$. Thus, each state can be represented by an integer, which we assume takes constant space. There are at most $O((d+2)^{\frac{1}{2}\log_{d+2}n}\,|\Sigma|)$ entries in the transition table for each leaf module. There are at most $O((d+2)^{\frac{1}{2}\log_{d+2}n}\,|\Sigma|\,h)$ entries in the transition table for internal modules, where h is the number of layers in that module. There are altogether $\leq \frac{2m}{\frac{1}{2}\log_{d+2}n}$ layers counting all the internal modules; thus, the total space needed is $O(n^{\frac{1}{2}} \cdot \frac{m}{\log_{d+2}n})$ (the first factor corresponds to the the number of entries in one layer, and the second factor corresponds to the number of layers), which equals $O(\frac{n^{\frac{1}{2}}m}{\log_{d+2}n})$. (We assume that $|\Sigma|$ is a small constant.) The time to compute an entry in the transition table is $O(\log_2(d+2))$; thus, the preprocessing time needed is $O(n^{\frac{1}{2}}m)$.

Now we show that the time complexity for scanning the text is $O(n + \frac{nm}{\log_{d+2}n})$. For every character scanned, the time spent in a leaf module is constant, and the time spent in an internal module is $O(p)$, where p is the number of pseudo-nodes contained in the module. Since the total number of pseudo-nodes counting all modules is the total number of modules - 1 = $O(m / \log_{d+2}n)$, the total time spent for scanning a character is $O(m / \log_{d+2}n)$. So the total time complexity is $O(n + \frac{nm}{\log_{d+2}n})$. We have the following theorem.

```
Input: a regular expression $R$, a text $A = a_1 a_2 a_3 ... a_n$, and the error bound $d$
Output: the ending positions of those approximate matches in $A$

begin
        build the NFA for $R$ using Thompson's construction;
        find the initial value of every node using recurrence 5.1;
        decompose the NFA hierarchically into modules;
        build transition tables $Next_1$ and $Next_2$ for each module;
        for every module encode the $E$ values to be its initial state;
        Let $M$ be the root module, and let $t$ be the number of its nodes;
        for $j = 1$ to $n$ do
                $Transition_1$ $(M, a_j, e_0)$;
                $Transition_2$ $(M, e_0)$;
                if $e_{t-1} \leq d$, then report a match at position $j$;
end

Procedure $Transition_1$ (Module $M$, input character $a$, input value $e_g$) ;
begin
        Let $I_s$ be the current state of $M$;
        copy $e_g$ to $e_0$ of $I_s$;
        let $h$ be the number of layers in $M$;
        for $k = 1$ to $h$ do
                for every pseudo-node $y$ in layer $k$ do
                        copy the $E$ value of the input node of $M_y$ to $e_y$ of $I_s$;
                $I_s = Next_1(I_s, a, k)$;
                for every $M_y$ in layer $k$ do
                        $Transition_1$ $(M_y, a, e_y)$;
                for every pseudo-node $y$ in layer $k$ do
                        copy the $E$ value of the output node of $M_y$ to $e_y$ of $I_s$;
end;

Procedure $Transition_2$ (Module $M$, input value $e_g$) ;
begin
        Let $I_s$ be the current state of $M$;
        copy $e_g$ to $e_0$ of $I_s$;
        let $h$ be the number of layers in $M$;
        for $k = 1$ to $h$ do
                for every pseudo-node $y$ in layer $k$ do
                        copy the $E$ value of the input node of $M_y$ to $e_y$ of $I_s$;
                $I_s = Next_2(I_s, k)$;
                for every $M_y$ in layer $k$ do
                        $Transition_2$ $(M_y, e_y)$;
                for every pseudo-node $y$ in layer $k$ do
                        copy the $E$ value of the output node of $M_y$ to $e_y$ of $I_s$;
end ;
```

Figure 6: Algorithm Approximate_Regular_Expression_Matching

Theorem 5.2: Given a regular expression of size m, a text of size n, and the number of errors allowed d, the approximate regular expression pattern matching problem can be solved in time $O(n + \frac{nm}{\log_{d+2} n})$ and space $O(\frac{n^{1/2} m}{\log_{d+2} n})$. □

References

[AG87]

Apostolico, A., and C. Guerra, "The longest common subsequence problem revisited," *Algorithmica*, **2** (1987), pp. 315–336.

[ADKF70]

Arlazarov, V. L., E. A. Dinic, M. A. Kronrod, and I. A. Faradzev, "On economic construction of the transitive closure of a directed graph," *Dokl. Acad. Nauk SSSR*, **194** (1970), pp. 487–488 (in Russian). English translation in *Soviet Math. Dokl.*, **11** (1975), pp. 1209–1210.

[GG88]

Galil Z., and R. Giancarlo, "Data structures and algorithms for approximate string matching," *Journal of Complexity*, **4** (1988), pp. 33–72.

[GP90]

Galil Z., and K. Park, "An improved algorithm for approximate string matching," *SIAM J. on Computing*, **19** (December 1990), pp. 989–999.

[Hi75]

Hirschberg, D. S., "A linear space algorithm for computing longest common subsequences," *Communications of the ACM*, **18** (1975), pp. 341–343.

[HS77]

Hunt, J. W., and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, 20 (1977), pp. 350–353.

[LV88]

Landau G. M., and U. Vishkin, "Fast string matching with k differences," *Journal of Computer and System Sciences*, **37** (1988), pp. 63–78.

[LV89]

Landau G. M., and U. Vishkin, "Fast parallel and serial approximate string matching," *Journal of Algorithms*, **10** (1989).

[MP80]

Masek, W. J., and M. S. Paterson, "A faster algorithm for computing string edit distances," *Journal of Computer and System Sciences*, **20** (1980), pp. 18–31.

[My86]

Myers, E. W., "An O(ND) difference algorithm and its variations," *Algorithmica*, **1** (1986), pp. 251–266.

[My88]

Myers, E. W., "A four-Russians algorithm for regular expression pattern matching," *Journal of the ACM*, to appear. Also, Technical Report TR-88-34, Department of Computer Science, University of Arizona (October 1988).

[MM89]

Myers, E. W., and W. Miller, "Approximate matching of regular expressions," *Bull. of*

Mathematical Biology, **51** (1989), pp. 5–37.

[NKY82]
Nakatsu, N., Y. Kambayashi, and S. Yajima, ''A longest common subsequence algorithm suitable for similar text string,'' *Acta Informatica*, **18** (1982), pp. 171–179.

[Se80]
Seller P. H., ''The theory and computations of evolutionary distances: Pattern recognition,'' *Journal of Algorithms*, **1** (1980), pp. 359–373.

[Th68]
Thompson, K., ''Regular expression search algorithm,'' *CACM*, **11** (June 1968), pp. 419–422.

[Uk85a]
Ukkonen E., ''Finding approximate patterns in strings,'' *Journal of Algorithms*, **6** (1985), pp. 132–137.

[Uk85b]
Ukkonen, E., ''Algorithms for approximate string matching,'' *Information and Control*, **64**, (1985), pp. 100–118.

[WF74]
Wagner, R. A., and M. J. Fischer, ''The string to string correction problem,'' *Journal of the ACM*, **21** (1974), pp. 168–173.

[WS78]
Wagner, R. A., and J. I. Seiferas, ''Correcting counter-automaton-recognizable languages,'' *SIAM J. on Computing*, **7** (1978), pp. 357–375.

[WMMM90]
Wu, S., U. Manber, E. W. Myers, and W. Miller, ''An *O(NP)* sequence comparison algorithm,'' *Information Processing Letters*, **35** (1990), pp. 317–323.

[WM91]
Wu, S., and U. Manber, ''Fast text searching with errors,'' submitted to *CACM*. Also, Technical Report TR-91-11, Department of Computer Science, University of Arizona (June 1991).