A Sub-Quadratic Algorithm for Approximate Regular Expression Matching

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

Department of Computer Science
University of Arizona
Tucson, AZ 85721

May 1992

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

ABSTRACT

The main result of this paper is an algorithm for approximate matching of a regular expression of size m in a text of size n in time $O(nm/\log_{d+2} n)$, where d is the number of allowed errors. This algorithm is the first $o(mn)$ algorithm for approximate matching to regular expressions.

1. Introduction

Let $A = a_1 a_2 a_3 ... a_n$ and $B = b_1 b_2 b_3 ... b_m$ 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, Uk85, WF74, WMMM90] is a partial list). The approximate string-matching problem is a similar problem, with the difference being that B is to be matched to any substring of A (that is, we want the match B *inside* A) rather than to all of A. 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 R of size m. We want to find all the substrings in A that are within edit distance d to strings that can be generated by the regular expression.

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. This assumption allows us to perform some operations on $O(\log n)$ bits in constant time. In practice, one can indeed perform operations on w bits, where w is the computer word size, in essentially one unit of time, and most algorithms implicitly assume

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

[2] Supported in part by NLM grant LM04960, and by NSF grant CCR-9002351.

so. Algorithms that utilize this ability (besides regular arithmetic operations and addressing) are sometimes called *4-Russian algorithms*, after a seminal paper [ADKF70] that used that technique for Boolean matrix multiplication. This is, however, a much more general technique, which can lead to very impressive speedups in practice. We call it the *bit-parallel technique*. In our opinion, this technique is underutilized in the theory of algorithms.

Algorithms for approximate regular expression matching with running time of $O(mn)$ have been given by [WS78] and [MM89]. Wu and Manber [WM92] presented another algorithm (which is part of the *agrep* package) that performs very fast in practice for small regular expressions. Masek and Paterson [MP80] used the bit-parallel technique to obtain an $O(mn/\log n)$ algorithm for finding the edit distance between two simple strings. Myers [My92] used the bit-parallel 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. We also developed simpler and faster algorithms for special types of regular expressions, called *limited regular expressions*, which are common in practice [WMM94]. In this paper we present a new algorithm for the approximate matching of regular expressions with a running time of $O(nm/\log_{d+2} n)$.

2. Preliminaries

Most string-matching algorithms operate by scanning the text, character by character, recording some information and looking for matches. 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 of the automaton, and a good fast traversal algorithm for the automaton.

The input to the problem 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 (i.e., insertions, deletions, and/or substitutions). 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 approach of dividing the pattern into parts and processing each part in constant time. 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 [My92]) such that the modules 'communicate' among themselves in a particular way. The state of each module is maintained during the scan of A with 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.

We start by briefly describing Thompson's method for constructing an NFA for a given regular expression (see also [Th68, HU79, ASU86]).

(1) For each symbol $a \in \Sigma$ in R, an NFA accepting 'a' consists of a start node θ and an accepting node ϕ with a transition a from θ to ϕ.

(2) Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t respectively, then the following three construction rules are used to construct NFA's for $N(s|t)$, $N(st)$, and $N(s*)$.

(2.1) $N(s|t)$: We add two new nodes: θ, a new start node with ε-transitions to the start nodes of $N(s)$ and $N(t)$ (which cease to be start nodes), and ϕ, the accepting node of $N(s|t)$, with ε-transitions from the accepting nodes of $N(s)$ and $N(t)$ (which we don't call accepting nodes anymore) to θ.

(2.2) $N(st)$: The start node of $N(s)$ becomes the start node of $N(st)$ and the accepting node of $N(t)$ becomes the accepting node of $N(st)$. The accepting node of $N(s)$ is merged with the start node of $N(t)$.

(2.3) $N(s*)$: Like (2.1) we add two nodes where θ is the new start node and ϕ is the new accepting node. There is a new ε-transition from the accepting node of $N(s)$ to the start node of $N(s)$, and a new ε-transition from θ to ϕ. Also, there is an ε-transition from θ to the start node of $N(s)$ and an ε-transition from the accepting node of $N(s)$ to ϕ. The transition from the accepting node of $N(s)$ to the start node of $N(s)$ is called a *back edge*.

The NFA $N(R)$ constructed by Thompson's algorithm has the following properties ([ASU86, HU79]):

1. The number of nodes in $N(R)$ is no more than twice the number of the symbols and operators in R.
2. $N(R)$ has one start node and one accepting node. The start node has no incoming edges and the accepting node has no outgoing edges.
3. Each node of $N(R)$ has at most two incoming edges and two outgoing edges, which implies that the number of edges is bounded by $O(|R|)$. If a node has two incoming edges, then both are ε edges.
4. Any loop-free path on $N(R)$ contains at most one back edge, because the underlying graph is a reducible graph (see [MM89] for a proof).

Let R be a regular expression, and hereafter let $M = N(R)$ denote the corresponding NFA constructed using Thompson's construction with start node θ and final node ϕ. We call a node whose incoming edge is labeled by a symbol from Σ, an *L-node*, and a node whose incoming edges are labeled ε, an ε-*node*. Because all edges into a node have the same label, we can consider each node to be labeled with this common incoming label. The start node of the machine, which has no incoming edges, is considered to be labeled with ε. We number the nodes by a topological order disregarding the back edges (which were formed by the closure operation). Let r_i denote the character corresponding to node i, and let $Pre(i)$ denote the predecessors of node i, namely, the nodes in M that have edges that point to node i. Let $\overline{Pre}(i) \subseteq Pre(i)$ denote the predecessors of i excluding back edges.

Next, we present a dynamic-programming type recurrence that is the basis of the algorithm. Let $E[i, j]$ be the minimum edit distance between any string that can reach node *i* when it is used as an input to *M* and any substring of *A* that ends at a_j. For a given *j*, the set of values $E[i,j]$ over all nodes *i* in *M* constitute the *state* of our scanning automaton, and we call $E[i, j]$ the edit distance of node *i* after scanning *j*. As will be seen momentarily the data-dependencies of the recurrence for $E[i, j]$ are such that the state after scanning *j* can be determined just from the state after scanning $j-1$.

The dynamic programming recurrence for $E[i, j]$ is given below. To avoid cyclic dependencies in the recurrence due to the back edges of the Kleene closure construction, we use two passes to compute the state after scanning *j* from the state after scanning $j-1$. The value of $E[i, j]$ after the first pass is denoted by $E'[i, j]$. We let $E[Pre(i), j]$ denote $\min_{k \in Pre(i)} E[k, j]$, that is, the minimum edit distance to any predecessor of *i*. $E'[Pre(i), j]$ is defined similarly with E' replacing *E*, and $E[\overline{Pre}(i), j]$ is defined similarly with $\overline{Pre}$ replacing *Pre*. The exact recurrence is as follows:

For $j \in [0, n]$:

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

For $i \ne \theta$:

$$E[i, 0] \;=\; \begin{cases} \min E[\overline{Pre}(i), 0] + 1 & \text{if } i \text{ is an } L-node \\ \min E[\overline{Pre}(i), 0] & \text{if } i \text{ is an } \varepsilon-node \end{cases}$$

For $j \in [1, n]$ and $i \ne \theta$:

$$E'[i, j] \;=\; \begin{cases} \min(\, E[i, j-1] + 1,\ E[Pre(i), j-1] + \delta_{r_i, a_j},\ E'[\overline{Pre}(i), j] + 1\,) & \text{if } i \text{ is an L}-\text{node} \\ E'[\overline{Pre}(i), j] & \text{if } i \text{ is an } \varepsilon-\text{node} \end{cases}$$

where $\delta_{a, b}$ is 0 if $a = b$ and 1 otherwise.

$$E[i, j] \;=\; \min(\, E'[Pre(i), j],\ E[\overline{Pre}(i), j]\,) \;+\; (1 \text{ if } i \text{ is an L}-\text{node}) \tag{2.1}$$

This recurrence is equivalent to Figure 6 in [MM89], and its proof follows from the discussion in [MM89]. We prove it here for completeness. First, however, we 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 obtained 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. A node *i* receives the best E' value from all its predecessors including those connected by back edges, and the best *E* value from all its regular forward predecessors. 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]) by property 4 of the NFA *M*. Figure 1 shows an example of computing $E[i, j]$ by using

Recurrence (2.1).

Theorem 1: Recurrence (2.1) correctly computes the edit distance $E[i, j]$.

Proof: The proof is by induction. Suppose that $E[i, j-1]$, for all nodes i in the NFA, has been correctly computed. We want to show that $E[i, j]$ computed by recurrence 2.1 is correct. Recall that we refer to $E[i, j]$ as edit distance of node i after scanning j, and to $E[i, j-1]$ as the edit distance after scanning $j-1$. The base case for $j = 0$ is handled by the initial conditions which are obviously correct as they set $E[i, 0]$ to the length of the shortest string that takes one from the start node of M to node i.

The value of $E[i, j]$ must be the value of $E[k, j-1]$ for some k plus the cost of editing a string s into a_j where s can range over all strings on paths from k to i in M. Without loss of generality, we can assume that (a) a_j is inserted and then the symbols of s are deleted, or (b) a_j is matched or substituted for the first symbol of s and then the rest of s is deleted. This follows because if any prefix of s were first deleted then this prefix would take us to some vertex h and it would follow that the value of $E[i, j]$ is that of $E[h, j-1]$ plus the cost of aligning a_j with the remaining suffix of s. Moreover, in case (a) we can assume with out loss of generality that k is an L-node by the same reasoning.

First we show that $E'[i, j]$ correctly reflects the best edit distance over all strings w that do not take us across a back edge of M. If i is an ε-node then $E'[i, j]$ must get its best value from a predecessor along an edge that is not a back edge, i.e., $E'[i, j] = E'[\overline{Pre}(i), j]$. Its value cannot come from $E[Pre(i), j-1]$ or $E[i, j-1]$ as either of these require i to be an L-node by our assumption about k. On the other hand if i is an L-node then just before reaching i either (i) the last symbol of w is deleted in which case $E'[i, j] = E'[\overline{Pre}(i), j]+1$, or (ii) a_j is deleted

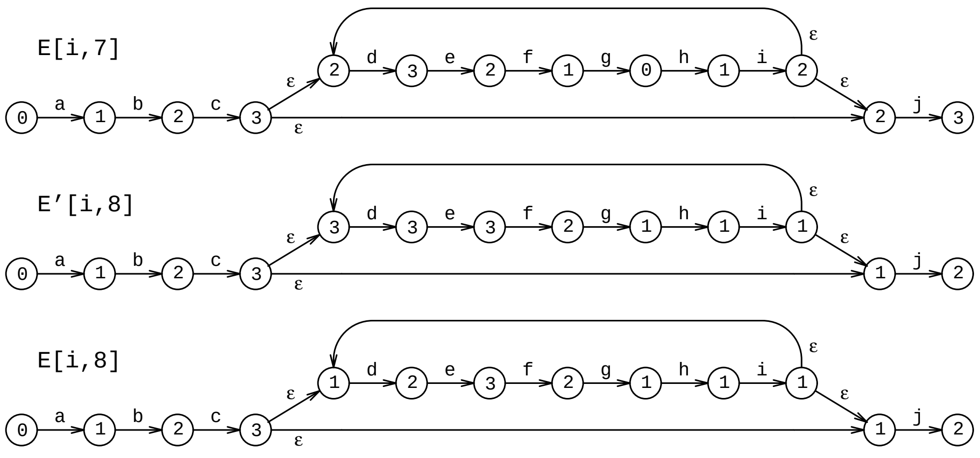

Figure 1: An example of computing $E[i, j]$ where $R = abc(defghi)^*j$ and $A = abcdefgi$.

(implying $w = \varepsilon$) in which case $E'[i, j] = E[i, j-1] + 1$, or (iii) r_i is substituted or matched to a_j (implying $w = r_i$) in which case $E'[i, j] = E[\underline{Pre}(i), j-1] + s_{i,j}$. Note that in case (i) we only consider the predecessors in $\overline{Pre}(i)$ as w is not permitted to take us across a back edge. Moreover, $E'[i, j]$ must be the minimum of one of the three cases and so the first pass of Recurrence (2.1) is correct.

It remains to enlarge the range of w to the set of strings that take us across at most one back edge. If the best value of $E[i, j]$ is obtained for a choice of w which does not involve a back edge then $E[i, j] = E'[i, j]$. Otherwise $E[i, j]$ obtains its value from a predecessor k's edit distance after scanning j. If k is in $\overline{Pre}(i)$ then clearly $E[i, j] = E[k, j] + (1$ if i is an L-node). However if the edge from k to i is a back edge then $E[k, j]$ must equal $E'[k, j]$ as the string minimizing it must traverse no back edges. So in this case $E[i, j] = E'[k, j] + (1$ if i is an L-node). Thus $E[i, j]$ cannot be less than the right hand side of the second pass of Recurrence (2.1) as this right hand side is less than all the terms above. Moreover, all of the terms in the recurrence reflect potential edit scripts and so equality holds. $\square$

3. The Algorithm

Recurrence (2.1) leads to an algorithm whose running time is $O(nm)$ in the worst case, because Thompson's construction guarantees that $|Pre(i)| \le 2$ for all i. We improve the running time in the following way. First, we allow only $d+2$ values for the $E[i,j]$'s: If the edit distance is d, then there is no need to distinguish among values $> d$, and they can be replaced by $d+1$. Secondly, we 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.

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 [My92], 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 the vertices of T as specified in Lemma 2 and illustrated at the left in Figure 2. Note that the partitioning of T is such that each block of vertices in the partition induces a subgraph that is a connected subtree of T. We describe such blocks as *connected*.

Lemma 2: For any $K \ge 2$, we can partition the parse tree T for a regular expression R into a connected block U that contains T's root and has no more than K vertices, and a set of other connected blocks, denoted by X, each having between $\lceil K/2 \rceil$ and K vertices.

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.

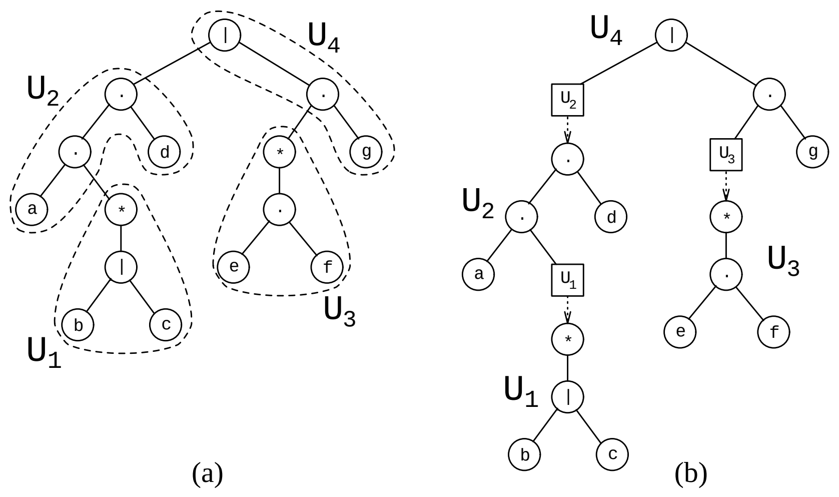

Figure 2: Parse tree of *a(b|c)*d|(ef)*g* and its decomposition with $K = 4$.

There are two cases:

(a) 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 vertices in U_c, and k_d be the number of vertices 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 $U = \{ r \}$.

(b) 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 vertices 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 is completed. □

Given the partitioning of Lemma 2, we connect the subtrees induced by the connected blocks in the following way. Let T_g and T_h be subtrees, and assume that T_g is connected to T_h by an edge (v, u) such that $v \in T_g$ and $u \in T_h$. We add a *pseudo-vertex* to T_g to represent u, and we call T_g a *parent* of T_h. This pseudo-vertex will serve to communicate values between parents and their children. We call the subtrees (with the extra pseudo-vertices) *modules*. We will use the term *original vertices* to indicate vertices that are in T (i.e., vertices that are not pseudo-vertices). The illustration at the right of Figure 2 shows the modules for the decomposition given at the left. A square denotes a pseudo-vertex and a circle an original vertex.

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. Specifically, each module of the NFA is the NFA corresponding to the regular expression modeled by a module of the parse tree where pseudo-vertices should be thought of as modeling a special symbol that matches any string denoted by the regular expression of the subtree it is connected to. Inside such an NFA module M_g, an edge corresponding to a pseudo-vertex is called a *pseudo-edge* and any other edge is termed *original*. If the subtree for module M_g is the parent of the subtree for module M_h, then M_g is called the *parent* of M_h and M_h is a *child* of M_g. Conceptually, the pseudo-edge in M_g that corresponds to M_h is to be thought of as representing the action of M_h (and all its children, recursively). A module that contains only original edges is called a *leaf* module. A module that contains pseudo-edges is called an *internal* module. The start state and final states of module M_h, denoted θ_h and ϕ_h are considered the *input* and *output* of the module. Moreover, within the parent module M_g let θ_h and ϕ_h denote the vertices at both ends of the pseudo-edge. Which usage of the notation is in effect will always be clear from context. It is not hard to see, based on Lemma 2, that for a given constant *K*, we can decompose the NFA into a collection of modules such that 1) each module contains not more than *K* nodes, and 2) the total number of modules is bounded by $O(m/K)$. Figure 3 shows the NFA decomposition for the regular expression and parse tree decomposition of Figure 2. Pseudo-edges are shown as dashed lines.

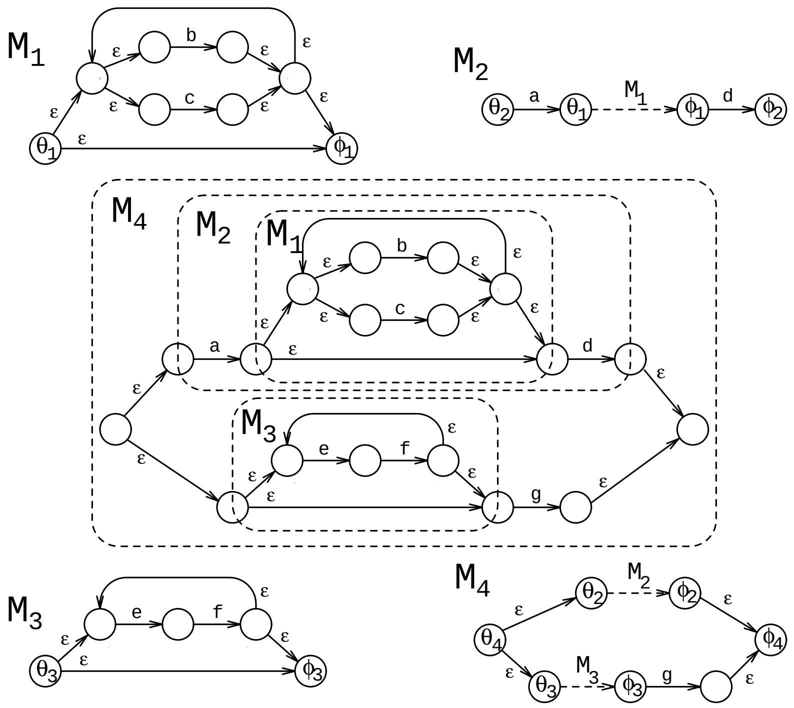

Figure 3: The NFA for *a(b|c)*d|(ef)*g* and its decomposition with $K = 4$.

As will be seen later in the analysis of the forthcoming algorithm the desired choice of module size is $K = \frac{1}{2}\log_{d+2} n - 1$. As the text is scanned by our algorithm we will need to determine the values $E[i, j]$ for every node of every module in the decomposition of M. Recall that we think of the set $\{ E[i, j] : i \in M \}$ as the state of our automaton after scanning a_j, and the basic $O(nm)$ algorithm realizes the transition function that takes M from one state to the next in $O(m)$ time by applying Recurrence (2.1) directly in two passes. We think of each of the two passes as inducing a state transition, the first depending on the scanned symbol, and the second an ε-transition independent of the symbol. For our bit-parallel algorithm, we need to precompute tables modeling these transition functions so that the state of a module can be advanced in $O(1)$ time. Consider a module M_g containing $t \le K$ nodes. A current state s of M_g is a vector of t values $(e_0, e_1, \ldots, e_{t-1})$, where $e_i \in [0, d+1]$ is the current 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. We write $I_s < i >$ to refer to the component in I_s that corresponds to e_i.

First, we consider the transition functions of a leaf module, M_g, that contains no pseudo-edges. In this case, we can implement the transition functions for M_g by precomputing two transition tables as follows. For every possible current state s represented by the integer I_s, every possible input character a, and every possible pass 1 value e of θ_g after scanning a, we precompute the state transition for M_g using pass 1 of recurrence 2.1. The result is encoded as a state integer, $I_{s'}$ and is stored in a transition table $Next_1[I_s, a, e]$. Precisely,

$$I_{s'} < \theta_g > \; = \; e$$
$$I_{s'} < i > \; = \begin{cases} \min(I_s < i > + 1,\ I_s < Pre(i) > + \delta_{a,\, r_i},\ I_{s'} < \overline{Pre}(i) > + 1) & \text{if } i \ne \theta_g \text{ is an L-node} \\ I_{s'} < \overline{Pre}(i) > & \text{if } i \ne \theta_g \text{ is an } \varepsilon\text{-node} \end{cases}$$

It is critical to note that the new pass 1 state value of M_g's start node θ_g must be given explicitly as its value can be a function of its predecessors in its parent module. Conversely, given e, the new state value of every other node in M_g is dependent only on the values encoded in I_s. Once the table $Next_1$ has been computed, we can advance a state of M_g to its pass 1 state after scanning a in constant time. The building of the pass 2 transition table is similar except that the input character a is not needed. Thus the pass 2 transition table, $I_{s''} = Next_2[I_{s'}, e]$ requires only a pass 1 state $I_{s'}$ and a pass 2 state value e of θ_g to produce the pass 2 state of the module. Precisely,

$$I_{s''} < \theta_g > \; = \; e$$
$$I_{s''} < i > \; = \; \min(I_{s'} < Pre(i) >,\ I_{s''} < \overline{Pre}(i) >) + (1 \text{ if } i \ne \theta_g \text{ is an L-node})$$

Having treated the simple case of leaf modules we generalize this treatment to the case of internal modules. Leaf modules will automatically be handled within this more general framework. Assume that $\theta_h \rightarrow \phi_h$ is a pseudo-edge for M_h in module M_g. Suppose that u and v are original nodes in M_g, and that u is a predecessor of θ_h and v is a successor of ϕ_h. Note that the computation of the next state value for the input node of M_h depends on the next state value of u, and the computation of the next state value for v depends on the next state value and potentially the current state value (if v is an L-node) of the output node of M_h. In other words, the computation of the next state value for v has to wait until the computation in module M_h has been completed. This dependency on pseudo-edges prohibits us from building a transition table that can be used to compute the state transition for M_g in constant time. Fortunately, as will become clear when we give the analysis of the algorithm, we do not have to implement the transition functions for an internal module in constant time. It will be sufficient to implement the state transitions for M_g in $p+1$ steps, where p is the number of children of M_g.

Consider partitioning the nodes of M_g into $p+1$ *layers* as follows. First number the nodes of M_g in topological order where all nodes of subautomaton $N(S)$ have a smaller number then all nodes of subautomaton $N(T)$ in an alternation subautomaton $N(S|T)$. Suppose that the $\theta_1 < \theta_2 < \cdots < \theta_p$ are the start states of the p children modules of M_g. Layer $h \in [0, p]$ of M_g is the set of nodes whose topological numbers are in the interval $[\phi_h, \theta_{h+1}]$ where we assume $\phi_0 = \theta_g$ and $\theta_{p+1} = \phi_g$ to simplify the definition. Note that in the topological numbering $\phi_h = \theta_h + 1$, so the layers do indeed partition the nodes of M_g. We say that child M_h *separates* layers $h-1$ and h. Figure 4 gives an example of a layer partitioning and the topological order used to define it. The set of edges $u \rightarrow v$ for which u is in one layer and v is in another satisfy the following key properties: (1) every edge in the set is either an ε-edge or a pseudo-edge, and (2) exactly one edge in the set is a psuedo-edge and it separates v's layer from u's layer.

For an internal module M_g with p children, we build $p+1$ pass 1 and pass 2 transition tables, one for each layer. Specifically, we compute tables $Next_1^h[I_s, a, e]$ and $Next_2^h[I_{s'}, e]$ that are specified exactly as given in (3.1) and (3.2) for a leaf module *except* that ϕ_h replaces θ_g everywhere. In other words, the next state value e is now for the final state of child module M_h if $h > 0$.

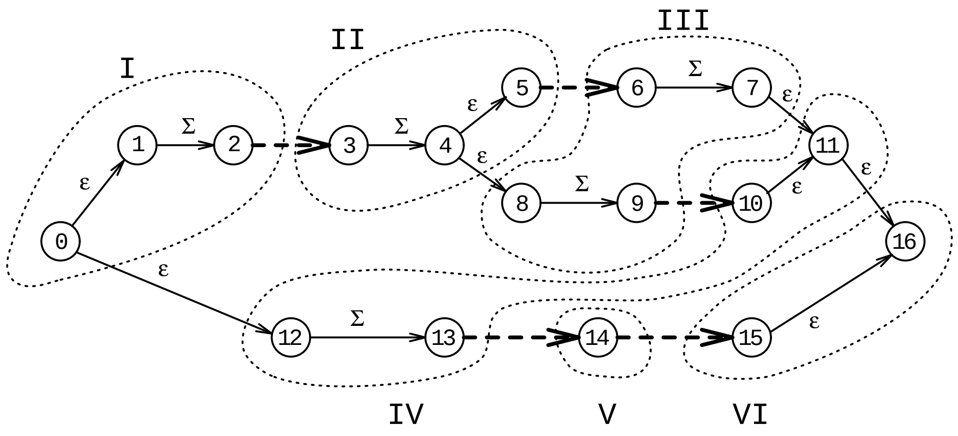

Figure 4: An example of a layer partitioning.

The tables only advance the state values of the nodes in the given layer. The state value of nodes encoded in integer I_s will be advanced in order of layer with intervening updates of the child separating the layers. Thus at the time one is just about to advance layer h the state values of $i \le \theta_h$ are those after the transition, and the state values of $i \ge \phi_h$ are those before the transition. It is then critical to observe that the nodes in layer h that have predecessors in another layer are ϕ_h and possibly a few ε-nodes. The next state value of the ε-nodes only depend on the next state value of their predecessors (and not the previous state value) by the form of Recurrence (2.1). Moreover, I_s still contains ϕ_h's previous state value and its next state value is provided explicitly by e. Thus the construction is such that all the values needed to advance the state of layer h is indeed encoded in I_s and e at the time the layer is advanced.

The complete algorithm is given in Figure 5. For each character of the text A, the state of the root module of the decomposition is advanced first by function $Transition_1$ and then function $Transition_2$ realizing the pass 1 and pass 2 transitions for every module recursively. Each function returns the current value of the module's final state after the transition so that the algorithm reports a match whenever a value not greater than d is returned by the invocation of $Transition_2$ on the root module. Focusing on $Transition_1(M, a, e)$, the function advances the state of M to the pass 1 state after scanning a assuming that e is the pass 1 state value after scanning a for M's start node θ. Note by Recurrence (2.1) that e is 0 for the root module. $Transition_1$ accomplishes this goal by advancing each layer of its state I in increasing order, recursively advancing the state of the child M_h separating layer $h-1$ from layer h, as it proceeds. Observe that after advancing the state over layer $h-1$ with a lookup in $Next_1^{h-1}$, the value of $I < \theta_h >$ is the next state value of thet start node of M_h needed for the parameter e in the recursive application of $Transition_1$ to M_h. Moreover, the recursive call to M_h returns the next state value of its final state, ϕ_h, and this is exactly the e value needed as input to the table lookup for layer h. In this way, the relevant next state values are chained together to cleanly handle the interdependence of the modules. Note that a leaf module has one layer and is just a special case of the more general treatment for internal modules. The function $Transition_2$ is analogous to $Transition_1$ save that $Next_2$ tables are used and the input character a is not needed.

Theorem 3: 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 $O(\frac{n\,m}{\log_{d+2} n})$ and space $O(\frac{n^{1/2}\,m}{\log_{d+2} n})$.

Proof: Let's start with space complexity. Recall that by construction every module has no more than K nodes, and that there are $X = O(m/K)$ modules. We choose K to be $\frac{1}{2}\log_{d+2} n - 1$. First, the maximum number of possible states for a module is $(d+2)^K = O(n)$ and so each state can be represented by an integer under our unit-cost RAM model. There are at most $O((d+2)^K\,|\Sigma|\,(d+2))$ entries in each pass 1 and pass 2 transition table. Moreover, the total number of tables is the sum of the number of layers over all modules. Charging the first layer of a module to itself and every other layer to the child that separates it, it follows that the total number

```
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 the approximate matches in $A$

begin
    Build the NFA for $R$ using Thompson's construction.
    Find the initial value of every node using recurrence 2.1.
    Decompose the NFA hierarchically into modules.
    Build transition tables $Next_1$ and $Next_2$ for each module.
    Encode the $E$ values of the initial states of all modules.
    Let $M$ be the root module.
    for $j \leftarrow 1$ to $n$ do
        $Transition_1 (M, a_j, 0)$
        if $Transition_2 (M, 0) \leq d$, then report a match at position $j$
end

Function $Transition_1$ (Module $M$, input character $a$, input value $e$) : integer
begin
    Let $I$ be the current state of $M$ and $p$ the number of pseudo-edges in $M$.
    for $h \leftarrow 0$ to $p - 1$ do
        $I = Next_1^h [I, a, e]$
        $e = Transition_1 (M_h, a, I < \theta_h >)$
    $I = Next_1^p [I, a, e]$.
    return $I < \phi >$
end

Function $Transition_2$ (Module $M$, input value $e$) : integer
begin
    Let $I$ be the current state of $M$ and $p$ the number of pseudo-edges in $M$.
    for $h \leftarrow 0$ to $p - 1$ do
        $I = Next_2^h [I, e]$
        $e = Transition_2 (M_h, I < \theta_h >)$
    $I = Next_2^p [I, e]$.
    return $I < \phi >$
end
```

Figure 5: The sublinear approximate regular expression matching algorithm.

of layers is $2X - 1 = O(m/K)$. Thus the total number of entries in all the tables is $O((d+2)^{K+1} m/K)$ (assuming $|\Sigma|$ is a constant) and since it takes $O(K)$ time to compute each entry it takes $O((d+2)^{K+1} m)$ time to compute these tables. Since $(d+2)^{K+1} = n^{1/2}$ by the choice of K, it follows that the tables occupy $O(n^{1/2} m/ \log_{d+2} n)$ space and $O(n^{1/2} m)$ preprocessing time is spent computing them.

Next we show that the time complexity for scanning the text is $O(nm/ \log_{d+2} n)$. For every character scanned, the time spent in a module is proportional to the number of layers in it. But we have already shown that the total number of layers is $O(m/K)$ and so this much time is taken scanning each character. Thus $O(nm/K) = O(nm/\log_{d+2} n)$ time is spent scanning the text. The only other cost is the $O(m)$ time it takes to produce the decomposition of M. Thus the scanning time is dominant and the proof is complete. $\square$

References

[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.

[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.

[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.

[My92]
Myers, E. W., ‘‘A four-Russians algorithm for regular expression pattern matching,’’ *J. of ACM* 39, 2 (1992), 430-448.

[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.

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

[Uk85]
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.

[WMM94]
Wu, S., U. Manber, and E. W. Myers, ‘‘A Sub-quadratic Algorithm for Approximate Limited Expression Matching,’’ Technical Report XXX, submitted for publication.

[WM92]
Wu S., and U. Manber, ‘‘Fast Text Searching Allowing Errors,’’ *Communications of the ACM* **35** (October 1992), pp. 83–91.