A Four Russians Algorithm for Regular Expression Pattern Matching

GENE MYERS

The University of Arizona, Tucson, Arizona

Abstract. Given a regular expression R of length P and a word A of length N, the membership problem is to determine if A is in the language denoted by R. An $O(PN/\lg N)$ time algorithm is presented that is based on a $\lg N$ speedup of the standard $O(PN)$ time simulation of R's non-deterministic finite automaton on A using a combination of the node-listing and ''Four Russians'' paradigms. This result places a new worst-case upper bound on regular expression pattern matching. Moreover, in practice the method provides an implementation that is faster than existing software for small regular expressions.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems - *Pattern matching*; I.1.2 [**Algebraic Manipulation**]: Algorithms - *Analysis of Algorithms*; I.5.0 [**Pattern Recognition**] General.

General Terms: Algorithms, Theory

Additional Keywords and Phrases: Finite Automaton, Four Russians Paradigm, Node listing, Regular Expression

---

This research was supported in part by the National Library of Medicine under Grant R01 LM4960.

Author's address: Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

0. *Introduction*

The ‘‘Four Russians’’ paradigm, first introduced in [AKD70], decomposes a problem into smaller subparts, precomputes in a table all possible outcomes of a small computation, and then solves the larger problem by looking up a series of solutions to its subparts. This technique led to an $O(N^3/lg\ N)$ time boolean matrix multiplication algorithm [AKD70, AHU75(§6.6)], and an $O(N^2/lg\ N)$ time sequence comparison algorithm [MaP80]. Both of these results assumed a RAM model with logarithmic operation costs, i.e. operations take time proportional to the size (number of bits) in their operands. In this paper we assume that operation costs are uniform. Precisely, a *lgN-bit uniform RAM* can add and compare integers of size $O(N)$, and can access an $O(N)$ memory, all in constant time. No assumptions are made about the complexity of multiplicative operations because we do not need them. Under this more powerful model the results above improve to $O(N^3/lg^2 N)$ and $O(N^2/lg^2 N)$, respectively.

We present an $O(PN/lg\ N)$ worst-case time and space algorithm for determining if a word A of length N is in the language denoted by a regular expression R of length P. For a small integer parameter K, our method achieves a K-fold speedup of the standard $O(PN)$ state-set simulation of R’s non-deterministic finite automaton, F, on the word A [Tho68, Sed83(§9)]. For a given K, F is hierarchically decomposed into $O(P/K)$ *modules*. In $O(K2^K)$ time and $O(2^K)$ space, tables are constructed that permit the state-set of a module to be advanced in $O(1)$ time. In essence, the tables encode a deterministic finite automaton for each subautomaton module of F and information about the connections between them. With these tables the module-based simulation of F on A can be performed in $O(PN/K)$ time. Thus, on a B-bit uniform RAM, we have an $O(P(N/B + 2^B))$ time, $O(P2^B/B)$ space algorithm for regular expression pattern matching. If $B \geq (lg\ N)/2$, then choosing $K = (lg\ N)/2$ yields an $O(PN/lg\ N)$ time and space algorithm. These results assume the alphabet Σ is finite, as does the result in [MaP80]. This restriction can be relaxed at an additional cost in the time complexity of $O(Nlg\ P)$.

Our result represents a worst-case improvement over the standard state-set simulation [Tho68], because it is an $O(lg\ P)$ factor faster, regardless of whether a uniform or logarithmic cost model is chosen. The standard $O(PN)$ time claim relies on a $lg\ P$-bit uniform model, in which case our algorithm takes $O(PN/lg\ P)$ time. Under the logarithmic cost model, the standard algorithm is $O(PNlg\ P)$ and ours is $O(PN)$. Thus our algorithm places a new upper bound on regular expression pattern matching and affirmatively answers an open "stringology" problem posed in [Gal85].

Unlike the implementation of the sequence comparison algorithm of [MaP80], which outperformed its standard counterpart [WaF74] only when $N \geq 262{,}000$ [MaP83], our method yields an implementation on architectures supporting bit-vectors that is faster than existing software for small patterns. Most machines support at least 16-bit word operations, so our strategy in practice is to choose $K = 14$ regardless of N or P. With this choice, the tables require less than $2520P$ words of memory, and the time to build them is less than $4P$ milliseconds on a VAX8650

running 4.3bsd UNIX. For small patterns, our implementation is competitive with *egrep*, which employs an $O(N)$ expected-time algorithm [Aho80, ASU85(§3.7)] and is the fastest tool currently used in practice.

1. *Regular Expressions, Parse Trees, and Finite Automata*

Regular expression are built up from individual symbols via union, concatenation, and concatenation-closure operations. Formally, the set of *regular expressions* over a given alphabet, Σ, is defined recursively as follows:

(1) If $a \in \Sigma \cup \{\varepsilon\}$, then a is a regular expression.

(2) If R and S are regular expressions, then so are $(R)|(S)$, $(R)(S)$, and $(R)*$.

A regular expression determines a *language*, i.e., a set of *words* where each word is a sequence of symbols from Σ. The precise language-defining rules are as follows. If $a \in \Sigma \cup \{\varepsilon\}$, then a denotes the set containing the single word a. The word ε is the unique word of length zero, and ε is assumed to be a symbol not in Σ. $(R)|(S)$ is the union of the languages denoted by R and S. $(R)(S)$ denotes the set of words obtained by concatenating a word in R and a word in S. Finally, $(R)*$ consists of all words that can be obtained by concatenating zero or more words from R. Unnecessary parentheses may be removed by observing that concatenation and union are associative, and by utilizing the ''natural'' precedence of the operators, which places * highest, concatenation next, and | last. For example, $abb|b|cb*$ denotes the language, $\{ abb, b, c, cb, cbb, \cdots \}$. Note that our definition is differs slightly from the usual one in that we exclude the empty language.

While regular expressions permit the convenient textual specification of *regular languages*, their parse trees better model their hierarchical structure. A *parse tree* T_R for a regular expression R consists of a labeled, rooted, and ordered tree with the following properties.

(1) The leaves are labeled with symbols from $\Sigma \cup \{\varepsilon\}$.

(2) The interior vertices are labeled with |, ·, or *.

(3) Interior vertices labeled | or · have two sons, those labeled * have one.

Regular parse trees correspond in a one-to-one fashion with regular expressions as shown by the following inductive construction. For the basis, regular expressions $a \in \Sigma \cup \{\varepsilon\}$ are modeled by parse trees consisting of a single leaf labeled a. Inductively, suppose expression R is modeled by tree T_R, and expression S is modeled by tree T_S. Then regular expression RS is modeled by a tree whose root is labeled · and whose sons are the roots of subtrees T_R and T_S. The regular expression $R|S$ is modeled by a tree whose root is labeled | and whose sons are the roots of subtrees T_R and T_S. Finally, $R*$ is modeled by a tree whose root is labeled * and whose son is the root of subtree T_R. Figure 1 gives a regular expression R and its corresponding parse tree T_R.

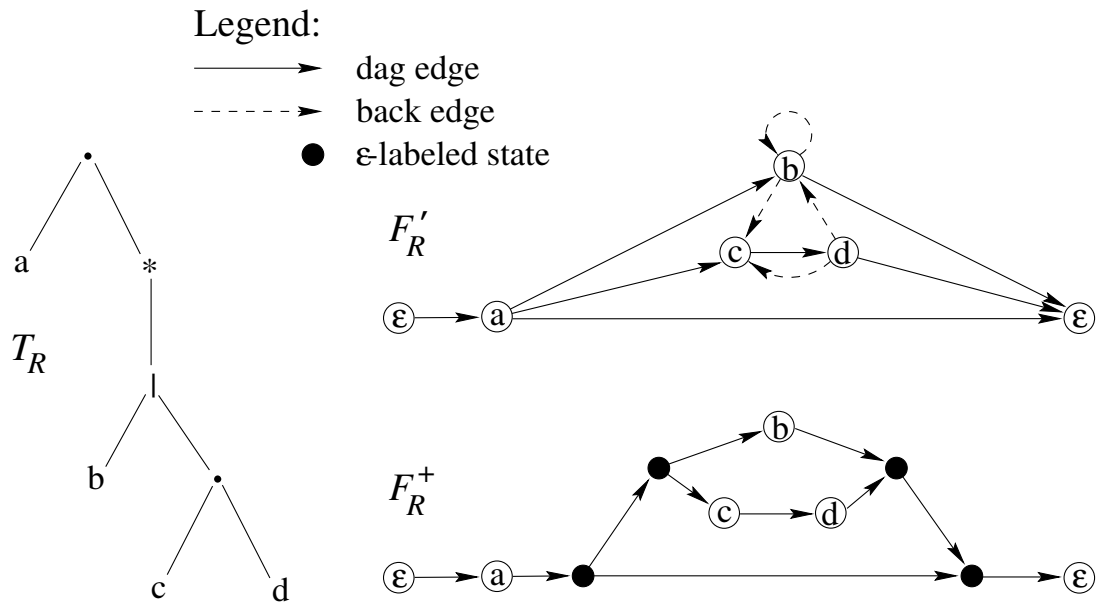


FIG. 1. T_R, F'_R, and F^+_R for $R = a\ (b|\ cd)^*$.

The finite state machine counterpart to a regular expression is best suited to the task of recognizing the words in a regular language. Several different models of finite automata have appeared, but all are equivalent in power and recognize exactly the class of regular languages. The complexity results of this paper require the use of a non-deterministic, ε-labeled model, and labeling states instead of edges yields simpler software. Formally, a *finite automaton*, $F = < V, E, \lambda, \theta, \phi >$, consists of:

(1) A set, V, of vertices, called *states*.

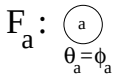
(2) A set, E, of directed edges between states.

(3) A function, λ, assigning a ''label'' $\lambda(s) \in \Sigma \cup \{\varepsilon\}$ to each state s.

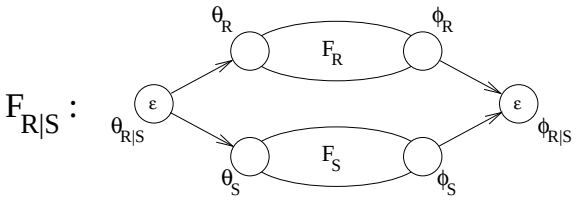
(4) A designated ''source'' state, θ, and a designated ''sink'' state, ϕ.

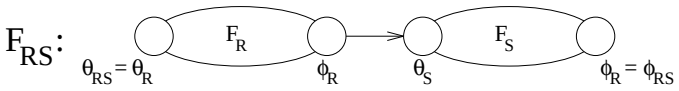
Intuitively, F is a vertex-labeled directed graph with distinguished source and sink vertices. A directed path through F *spells* the word obtained by concatenating the state labels along the path. Recall that ε acts as the identity element for concatenation, i.e., $v\varepsilon w = vw$. So, one may think of spelling just the non-ε labels on the path. $L_F(s)$, the *language accepted at* $s \in V$, is the set of words spelled on paths from θ to s. The *language accepted by* F is $L_F(\phi)$.

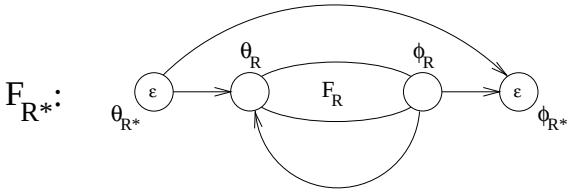
For any regular expression, R, the following recursive method constructs a finite automaton, F_R, that accepts exactly the language denoted by R. For $a \in \Sigma \cup \{\varepsilon\}$, construct the automaton:



Suppose that R is a regular expression and automaton F_R with source θ_R and sink ϕ_R has been constructed. Further suppose that automaton F_S with source θ_S and sink ϕ_S has been constructed for regular expression S. Then automata for $R|S$, RS, and $R*$ are constructed as follows:







Each diagram precisely specifies how to build the composite automaton from F_R and F_S. For example, F_{RS} is obtained by adding an edge from ϕ_R to θ_S, designating θ_R as its source state, and ϕ_S as its sink.

A straightforward induction shows that automata constructed for regular expressions by the above process have the following properties:

(1) The in-degree of θ is 0.

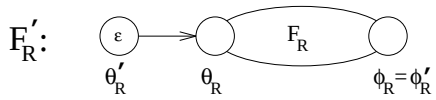
(2) The out-degree of ϕ is 0.

(3) Every state has an in-degree and an out-degree of 2 or less.

(4) $|V| < 2P$ where $P = |R|$, i.e., the number of states in F_R is less than twice R's length.

For the sub-automaton to be used in the subsequent modular decomposition, it is required that the start and final states of the automaton be distinct and ε-labeled. This alteration is simply accommodated by modifying the

automaton F_R produced for R above as follows:



The altered automaton F'_R satisfies the properties above, save that now $|V| \leq 2P+1$. The key observation is that for any regular expression R there is a non-deterministic, ε-labeled automaton whose size, measured in vertices or edges, is linear in the length of R. Figure 1 gives a regular expression R and its finite automaton F'_R. Hereafter, the subscript R is dropped whenever it can be inferred from context.

The algorithms to be developed depend on another, more subtle, property of the constructed automata. Call the constructed edges that run left-to-right in the above pictures *dag edges*. The remaining *back edges* consist of just those from ϕ_R to θ_R in the diagram of F_{R*}. For those familiar with the data flow analysis literature, Part 2 of Lemma 1 asserts that the *loop connectedness parameter* of the automaton's graph is 1 [HeU75].

LEMMA 1. *Consider any finite automaton, F, constructed by the above processes. (1) Any cycle-free path in F that begins at* θ *consists solely of dag edges. (2) Any cycle-free path in F has at most one back edge.*

PROOF. Suppose a path from θ has a back edge. It must connect the final state of some sub-automaton, *S*, to its start state. But any path passing to the final state of *S* must enter via *S*'s start state by the hierarchical construction of *F*. Thus the back edge forms a cycle. This proves part 1 by contradiction.

Let *p* be a cycle-free path in *F* and suppose that, contrary to part 2, *p* contains two back edges. The first back edge in *p* connects the final state of some sub-automaton, *S*, to its start state. Consider the suffix, *q*, of *p* that follows that back edge. If *q* leaves *S*, then a cycle would be formed when it exits via *S*'s final state. But since *p* is cycle-free, *q* must stay within *F*. However, part 1, when applied to *S*, implies that *q* contains no back edges, contradicting the assumption that *p* contains two back edges. □

Our subsequent algorithms require an automaton F^+_R accepting the language denoted by R whose only ε-labeled states are the start and final states of the machine. We term such automata ε-*free* and for these automata each interior state of a path spells a symbol in Σ. F^+ is obtained from F' as follows. The vertices of F^+ are θ', ϕ', and all Σ-labeled states of F'. A path is an ε-*path* if it consists of one or more edges and its interior vertices are all labeled ε. There is an edge from *s* to *t* in F^+ if and only if there is an ε-path from *s* to *t* in F'. F^+ has $O(P)$ states, $O(P^2)$ edges, accepts the same language as F', and can be constructed from F' in $O(P^2)$ time. Figure 1 gives a regular expression R and its finite automaton F^+_R.

For each edge $s \rightarrow t$ of F^+ there is, by definition, at least one ε-path from *s* to *t* in F'. Choose an ε-path from *s* to *t* that has the fewest number of edges as the *image*, $\mu(s \rightarrow t)$, of edge $s \rightarrow t$. Note that $\mu(s \rightarrow t)$ must be cycle-free as

paths with cycles can be shortened by excising the cycle. Consequently, $\mu(s \rightarrow t)$ has at most one back edge by Lemma 1. Term an edge $s \rightarrow t$ of F^+ a *dag edge* if and only if its image consists solely of dag edges in F'. Otherwise, classify the edge as a *back edge* and note that its image has exactly one back edge in F'. Further observe that for both F' and F^+, the subgraph obtained by removing all back edges is acyclic.

Lemma 2 generalizes Lemma 1 for F^+ where a ''cycle-free'' path is considered to be one whose image in F' is cycle-free. A *shortcut* for a path $p = s \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow t$ is a path $s \rightarrow t_{i_1} \rightarrow \cdots \rightarrow t_{i_m} \rightarrow t$ such that $t_{i_1} t_{i_2} \cdots t_{i_m}$ is a subsequence of $t_1 t_2 \cdots t_n$, i.e., $i_k \in [1, n]$ and $i_{k-1} < i_k$, for all k. A shortcut of p is *proper* if it is not p itself. Note that since t_i does not necessarily have an edge to t_j for $j > i+1$, not all subsequences of the interior vertices of a path form a shortcut.

LEMMA 2. *Consider a path p in an ε-free automaton, F^+, constructed as above. (1) If p starts at θ then there is a shortcut of p consisting solely of dag edges. (2) There is a shortcut of p that has at most one back edge.*

PROOF. It suffices to show that if p itself does not satisfy (1) or (2), then there is a proper shortcut of p. This suffices because if we can repeatedly shorten shortcuts of p not satisfying (1) or (2), then we must eventually arrive at one that does. First observe that if p contains a cycle, truncating it produces a proper shortcut of p. So suppose p is cycle-free, does not satisfy (1) or (2), and contains the edges, $t_0 \rightarrow t_1 \rightarrow \cdots \rightarrow t_n$, where $t_0 = s$ and $t_n = t$. Let the image of p in F' be the path obtained by replacing each edge of p with its image, i.e., $\mu(p) = \mu(t_0 \rightarrow t_1) \cdot \mu(t_1 \rightarrow t_2) \cdot \cdots \cdot \mu(t_{n-1} \rightarrow t_n)$. Since p does not satisfy (1) or (2), it follows by Lemma 1 that $\mu(p)$ contains a cycle. Suppose v is the first state to appear twice in $\mu(p)$'s state list. There is an edge $t_{k-1} \rightarrow t_k$ such that the first occurrence of v is one of the vertices in $\mu(t_{k-1} \rightarrow t_k)$ excepting t_k. Similarly, there is an edge $t_{h-1} \rightarrow t_h$ for which the second occurrence is one of the vertices in $\mu(t_{h-1} \rightarrow t_h)$ excepting t_{h-1}. Of course, $k \leq h$, but also $k \neq h$ as an edge's image does not contain a cycle. Moreover, the edge $t_{k-1} \rightarrow t_h$ exists in F^+ because there is an ε-path or empty path from t_{k-1} to v and another from v to t_h. v can coincide with t_{k-1} or t_h but not both as p is cycle-free implies $t_{k-1} \neq t_h$. Hence there is indeed an ε-path from t_{k-1} to t_h. Thus we may delete states $t_k \cdots t_{h-1}$ from p's state list to obtain a proper shortcut of p. □

2. *The Modular Decomposition of R's Parse Tree*

Our modular decomposition of a finite automaton for R takes advantage of the hierarchical (inductive) formation of regular expressions. For this reason, it is easiest to first express our decomposition in terms of a decomposition of the associated parse tree, T_R, for R. As per convention established earlier, T will refer to T_R whenever R can be inferred from context.

Suppose T has L leaves and note that $L \leq P$. For a small integer K, it is possible to partition the edges of T into $O(P/K)$ sets such that the subgraph of T induced by each edge set is a parse tree, V, whose leaves may have interior

labels. That is, V is weakly connected and if a vertex has one son, it has all of its sons. Each edge set and its subgraph V is termed a *module*. Each module is guaranteed to have $\Theta(K)$ leaves except for the one containing the root of T which has $O(K)$ leaves. Figure 2 gives an example of our decomposition of the parse tree for $(a|\ bc)^* |\ (ab(c|\ d))^*$ with $K = 3$.

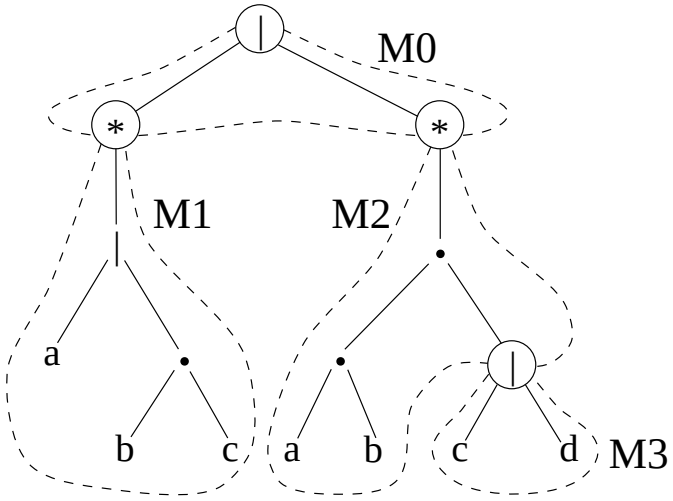


FIG. 2. A $K = 3$ partition of T_R for $R = (a|\ bc)^* |\ (ab(c|\ d))^*$.

LEMMA 3. *For $K \geq 2$, T can be partitioned into a module U that contains T's root and has no more than K leaves, and a collection X of modules that have between $\lfloor K/2 \rfloor + 1$ and K leaves each.*

PROOF. By induction on the size of the tree. For the basis, T consists of a single leaf. The hypothesis is true with X empty and $U = T$. For the induction, there are two cases depending on the outdegree of T's root, r.

First, suppose that r has two children, c and d, and let T_c and T_d be the subtrees rooted at c and d, respectively. Assume inductively that T_c can be partitioned into modules $\{ U_c \} \cup X_c$ satisfying the hypothesis, and similarly for T_d. Let $k_c \leq K$ be the number of leaves in U_c. If $k_c + k_d \leq K$ then let $X = X_c \cup X_d$ and let $U = \{ r \rightarrow c, r \rightarrow d \} \cup U_c \cup U_d$. Otherwise suppose without loss of generality that $k_c \geq k_d$. This implies that $k_c \geq \lfloor K/2 \rfloor + 1$. If $k_d + 1 \leq K$ then let $X = X_c \cup X_d \cup \{ U_c \}$ and let $U = \{ r \rightarrow c, r \rightarrow d \} \cup U_d$. Otherwise it must be that $k_d = k_c = K$ and one can let $X = X_c \cup X_d \cup \{ U_c, U_d \}$ and $U = \{ r \rightarrow c, r \rightarrow d \}$.

Finally, suppose that r has one child c and that T_c can be partitioned into module U_c and set of modules X_c. The hypothesis is made true by selecting $X = X_c$ and $U = \{ r \rightarrow c \} \cup U_c$. □

Now we show that the number of modules, M, in a K-partition of T is $\Theta(L/K)$ where L is the number of leaves in T. Suppose for the moment that every module has t leaves. Then M is the number of interior vertices in a t-ary tree with L leaves. It thus follows that $L = (t-1)M + 1$. In the best case, every module is of size K, in which case $M = \frac{L-1}{K-1}$. In the worst case every module is of size $\lfloor K/2 \rfloor + 1$, in which case $M = \frac{L-1}{\lfloor K/2 \rfloor}$. Now consider

partitions in which module sizes are not necessarily uniform. The smallest value of M is obtained when every module has K leaves except one which has between 1 and K. In this case, $M \geq \lceil \frac{L-1}{K-1} \rceil$. The largest value of M is obtained when (a) $L \leq K$ and there is one module, or (b) $L > K$ and every module has $\lfloor K/2 \rfloor + 1$ leaves except one which has at least two leaves. In either case (a) or (b), $M \leq \lceil \frac{L}{\lfloor K/2 \rfloor} \rceil$. Thus the bounds on the number of leaves in each module guarantee that M is $\Theta(L/K)$. Moreover, because L is $O(P)$, it follows that M is $O(P/K)$.

The language of regular expressions over alphabet Σ is $LL(1)$ and so a simple recursive-descent parser can convert R to T in $\Theta(P)$ time and space. Moreover, the induction argument of Lemma 3 immediately permits one to mark the root of each subtree in the partition with a simple bottom-up traversal of T. Since T is produced by the parser in post-order as it scans R, both T and its partition can be computed in a single $\Theta(P)$ left-to-right scan of R.

3. *The Modular Decomposition of R's Automaton*

Consider a module V of the partition of T produced by Lemma 3. Term a leaf of V *atomic* if and only if it is a leaf of T that is not labeled with ε. Note that the label of an atomic leaf is a symbol in Σ. A *modular* leaf of V is one that is not a leaf of T. Note that a modular leaf is labeled with an operator symbol and must coincide in T with the root of another module W of the partition. In this case, we say that V is the *father* of W, and, conversely, that W is a *son* of V. Let $\sigma_V \notin \Sigma$ be a unique symbol associated with each module V. Consider V when each modular leaf is relabeled with σ_W where W's root coincides in T with the leaf. This module is a parse tree for a regular expression, R_V, over the alphabet $\Sigma \cup \{ \sigma_W : W \text{ is a son of } V \}$. Let E_V be the automaton $F^+{}_{R_V}$ given by the construction in Section 1. Term a state of E_V *atomic* if it is labeled with a symbol from Σ, and term it *modular* if it is labeled with σ_W for some son W. Note that the modular and atomic leaves of V and the modular and atomic states of E_V are in one-to-one correspondence. Moreover, the construction of E_V guarantees that its start and final states, θ_V and ϕ_V, are distinct, ε-labeled, and consequently, neither atomic or modular.

Since regular languages are closed under the substitution of regular languages, the automata for the modules partitioning T can be hierarchically combined to form a finite automaton F^* for R. More precisely, for each module V, we recursively construct an automaton F_V accepting the language denoted by the regular expression for T_V, the subtree of T rooted at module V's root. For the base case where V has no sons, we let F_V be E_V. Now suppose V has $h \geq 1$ modular leaves and we have recursively constructed F_W for each of the h sons, W, of V. Then F_V is constructed by replacing each modular state labeled σ_W in E_V with the automaton F_W. Every edge into the modular state becomes an edge into the start state θ_W of F_W and every edge out of the modular state becomes an edge out of the final state ϕ_W of F_W. Thus, if U is the module containing T's root, then the automaton $F^* = F_U$ accepts the language denoted by R. F^* is called the *augmented* automaton for R with respect to the partition of T. Figure 3

gives F^* and its partition for the example of Figure 2. F^* has M module start states θ_V, M module final states ϕ_V, and L atomic states. Thus F^* has $O(P/K)$ ε-labeled states and $O(P)$ Σ-labeled states. Moreover, F^* has $O(PK)$ edges, $O(K^2)$ in each module. Constructing F^* is easily accomplished in $O(PK)$ time by building F_V for each module V and then redirecting the edges adjacent to modular states.

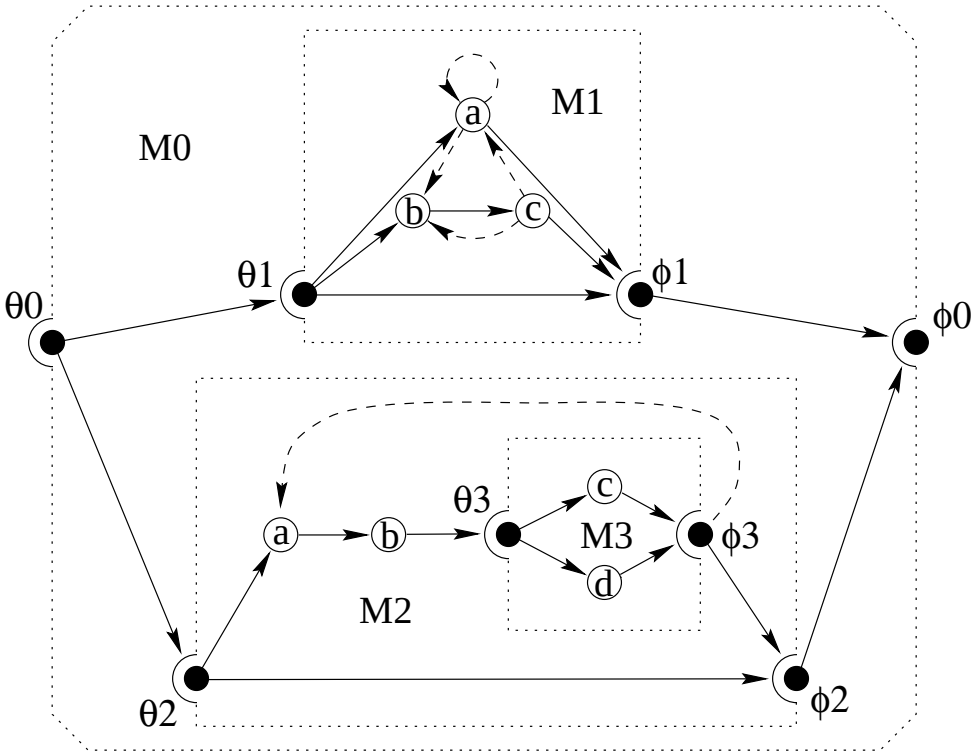


FIG. 3. F^* for the partition of Figure 2.

We now associate a set of edges and states of F^* with each module V in a manner that partitions F^*. The edges of a module V are those edges in F^* that were introduced by E_V. The edge sets partition the edges of F^* because E_V contributes exactly one copy of its edges to F^*. Note that all edges out of a state are in the same edge set, as are all edges into a state. It is thus sound to define a state s as belonging to that module $Mod(s)$ containing its outedges. With this definition, the set of states, Π_V, belonging to a module V is $\{ \theta_V \} \cup \{ \phi_W : W \text{ is a son of } V \} \cup \{ s : s \text{ is an atomic state of } E_V \}$. Note that the sets Π_V partition the set of all states of F^* except for its final state. In symmetry with $Mod(s)$, let $Pre(s)$ be the module containing the edges into s. From the construction of F^* it follows that the set of states, s, for which $Pre(s)$ equals V is $\{ \phi_V \} \cup \{ \theta_W : W \text{ is a son of } V \} \cup \{ s : s \text{ is an atomic state of } E_V \}$. Note that $t \rightarrow s$ implies that $t \in \Pi_{Pre(s)}$, i.e., the predecessors of s form a subset of the states belonging to module $Pre(s)$.

Consider classifying the edges of F^* as dag or back edges according to their type in the ε-free automaton E_V contributing the edge. Then Lemma 2 holds for F^*. This is most intuitively seen by consider the automaton F''

obtained by using F'_{R_V} instead of $F^+_{R_V}$ in the construction of F^*. The hierarchical construction of F'' by the same processes that constructed each F' implies that it satisfies Lemma 1. But then F^* is an ε-path reduction of F'' and the argument of Lemma 2 applies to F^*. This property of F^* is important to our algorithm as embodied in Corollary 4. The notation $s \rightarrow^* t$ asserts that there is a path from s to t all of whose vertices are labeled ε except possibly s.

COROLLARY 4. *If $s \rightarrow^* t$ in F^* then $s \rightarrow^* t$ along a path with at most one back edge, and none if $s = \theta$.*

PROOF. Simply note that shortcuts of a path spelling ε also spell ε and then apply Lemma 2. □

4. *The (K+1)-bit Uniform RAM Algorithm*

Our algorithm consists of a phase in which the regular expression is preprocessed into a number of tables, followed by a scanning phase in which the text is searched for occurrences of the pattern. The preprocessing phase consists of building F^* as described in Sections 1 through 3, and then constructing tables as detailed in Section 4.1 below. A *Goto* table models a deterministic finite automaton for each module, and a *Reach* table permits ε-paths between modules to be easily traversed. The scanning phase described in Section 4.2 mimics the standard state-set simulation of F^* save that the state-set of each module is modeled as an integer and advanced in $O(1)$ time with the *Goto* table. The subtle difficulty is that ε-labeled states reachable from states in other modules must be detected by an additional two-sweep closure pass over the modules. Nonetheless, the phase consumes $O(P/K)$ time per symbol scanned and thus gives a factor of K speedup over the standard state-set simulation. The treatment assumes Σ is finite and small. In Section 4.3 we present a refinement that circumvents this restriction at effectively no increase in space and time.

4.1 PREPROCESSING STEP. Our algorithm uses a $(K+1)$-bit integer to encode a subset of Π_V for each module V. Each state s in Π_V is assigned a unique bit position, $Bit(s)$, between 0 and $|\Pi_V| - 1 \leq K$. For $X \subseteq \Pi_V$, let the collection of set bits, $BIT(X)$, equal $\{ Bit(s) : s \in X \}$. For $Y \subseteq [0, K]$, let the integer, $NUM(Y)$, corresponding to set Y equal $\sum_{k \in Y} 2^k$. Note that NUM is an isomorphism, so its inverse exists. Finally, let the integer encoding $X \subseteq \Pi_V$ be $INT(X) = NUM(BIT(X))$.

For each module, a number of tables are built in the preprocessing step. $Setbit[k, Z]$ gives the integer resulting when the k-th bit of Z is set. It is needed because $O(1)$ integer multiplication is not assumed. $Reach[s, Z]$ is a boolean table whose entries are true if and only if there is a state in the set encoded by Z that is a predecessor of s. For this table, s ranges over just those states that are labeled with ε, i.e., the θ and φ states of F^*. *Reach* is used to trace subpaths spelling ε. Finally, $Goto[V, Z, a]$ gives the integer encoding the set of states in Π_V labeled a that are

successors of states in the set encoded by Z. *Goto* is used to discover paths ending with a specific symbol. A formal specification of the domain and entries of each table is given in the list below.

(1) For $k \in [0, K]$ and $Z \in [0, 2^{K+1}-1]$:

$$Setbit[\, k, Z] = NUM(\, NUM^{-1}(Z) \cup \{k\})$$

(2) For s an ε-labeled state and $X \subseteq \Pi_{Pre(s)}$:

$$Reach[\, s, INT(X)] = \text{There exists } t \in X \text{ s.t. } t \rightarrow s$$

(3) For V a module, $X \subseteq \Pi_V$, and $a \in \Sigma$:

$$Goto[\, V, INT(X), a] = INT(\, \{\, s : \text{There exists } t \in X \text{ s.t. } t \rightarrow s \text{ and } \lambda(s) = a \,\}\,)$$

Note that $t \in \Pi_V$, $t \rightarrow s$, and s atomic implies that $s \in \Pi_V$ by the construction of F^*. Thus *Goto* is properly defined since the set on the right is a subset of Π_V.

For a given value of Z, the entries, $Setbit[k, Z]$ for all k, can be computed in $O(K)$ integer steps. Thus all of *Setbit* can be computed in $O(K2^K)$ time and space. The $M2^{K+2}$ entries of the table *Reach* can be computed in $O(P2^K/K)$ time and space using the recurrence:

$$Reach[s, Z] \equiv \textbf{if } Z = 0 \textbf{ then } false \textbf{ else } Reach[s, Z - 2^k] \text{ or } t \rightarrow s \in F^+$$

where 2^k is the greatest power of 2 less than Z (i.e., $k = \lfloor lg\ Z \rfloor$), and t is the unique vertex for which $t \in \Pi_{Pre(s)}$ and $Bit(t) = k$. For a given ε-labeled state s, successive entries, $Reach[s, Z]$, can be computed in $O(1)$ time in increasing order of Z using the recurrence. The *Goto* tables can be efficiently computed using the same recursive decomposition over Z used for *Reach*. In this case the recurrence for positive Z is:

$$Goto[V, Z > 0, a] \equiv NUM(\, NUM^{-1}(Goto[V, Z - 2^k, a]) \cup BIT(Succ(t, a))\,)$$

where $k = \lfloor lg\ Z \rfloor$, t is the unique vertex for which $t \in \Pi_V$ and $Bit(t) = k$, and $Succ(t, a) = \{\, s : t \rightarrow s \text{ and } \lambda(s) = a \,\}$. Given a specific V, one first computes the *Succ* sets and then applies the recurrence for each a, and each Z in increasing order. For the $K|\Sigma|$ choices of t and a, the lists $Succ(t, a)$ can be built in $O(K|\Sigma|)$ time and space by setting the lists to empty and then for each $s \in \Pi_V$, for each t such that $t \rightarrow s \in F_V$, adding s to $Succ(t, \lambda(s))$. To perform the union in the recurrence, each element of $Succ(t, a)$ is added to $Goto[V, Z - 2^k, a]$ with a table lookup into *Setbit*. For a given V and Z, a total of $O(K)$ bits are set over all choices of a. Thus, the $M2^{K+1}|\Sigma|$ entries of *Goto* can be computed in $O(M2^K(|\Sigma| + K))$ time. Assuming $|\Sigma|$ is a constant, it takes $O(P2^K)$ time and $O(P2^K/K)$ space to build *Goto* and this dominates the complexity of the preprocessing step.

4.2 SCANNING STEP. The standard state-set simulation of an automaton reads the symbols of A from left-to-right while computing the set of states reachable from θ on paths spelling the prefix of A scanned thus far. More formally, suppose $A = a_1 a_2 \cdots a_n$ and let A^i denote the i-symbol prefix of A, i.e., $A^0 = \varepsilon$ and $A^i = a_1 a_2 \cdots a_i$ for $i > 0$. Further, let $\theta \underset{w}{\rightarrow} s$ assert that there is a path from θ to s spelling w. After reading the i'th symbol of A, the current set of states computed by the standard simulation is $S^i = \{ s : \theta \underset{A^i}{\rightarrow} s \}$. The standard algorithm computes S^0 from scratch, then for each i it computes S^i from S^{i-1} as it reads a_i, and it concludes by reporting that A matches R if and only if $\phi \in S^n$. Computing each current state-set takes $O(P)$ time for an automaton such as F'_R, giving an $O(PN)$ time, $O(P)$ space algorithm [Tho68, Sed83(§9)].

The algorithm of Figure 4 computes the same sequence of state-sets as the standard algorithm but in only $O(P/K)$ time per state-set. It achieves this speed up by modeling the current state-set as an array, $Set[V]$, of M $(K+1)$-bit integers, one for each module V. After scanning the i'th symbol of A, $Set[V]$ encodes the subset of S^i belong to V, i.e., $Set[V] = INT(\Pi_V \cap S^i)$. Note that $\underset{module\ V}{\cup} (\Pi_V \cap S^i) = S^i - \{\phi\}$ because the sets Π_V partition the states of F^* less ϕ. Thus the array Set does indeed model the current state-set of the standard simulation with the exception of ϕ whose omission is harmless since it has no successors in F^*.

Given the array Set modeling S^{i-1}, our algorithm reads a_i and updates Set so that it models S^i and does so in two $O(P/K)$ time phases. In the first *module phase*, Set is updated in lines 6 and 7 of Figure 4 by reseting each integer $Set[V]$ to $Goto[V, Set[V], a_i]$. After this phase, Set models the set of states , $\bar{S}^i$, that are atomic and in S^i. The proof is as follows. The states in $\bar{S}^i$ must all be labeled a_i and so $\bar{S}^i = S^i \cap L_{a_i}$ where $L_a = \{ s : \lambda(s) = a \}$. Moreover, $\Pi_V \cap S^i \cap L_{a_i} = \{ s : s \in \Pi_V$ and there exists $t \in S^{i-1}$ s.t. $t \rightarrow s$ and $\lambda(s) = a_i \}$. But by the construction of F^*, $t \rightarrow s$ and s atomic implies that t and s belong to the same module. Thus, $\Pi_V \cap S^i \cap L_{a_i} = \{ s :$ There exists $t \in \Pi_V \cap S^{i-1}$ s.t. $t \rightarrow s$ and $\lambda(s) = a_i \}$. It then follows that $INT(\Pi_V \cap S^i \cap L_{a_i}) = Goto[V, INT(\Pi_V \cap S^{i-1}), a_i]$. Thus after the module phase, $Set[V] = INT(\Pi_V \cap S^i \cap L_{a_i})$ and so models $\bar{S}^i$.

In the second *closure phase* of lines 8 and 9, the ε-labeled states of S^i are added to the state set $\bar{S}^i$ modeled by Set. Let χ be the set of ε-labeled states in F^* except for its start and final states. Note that χ has only $O(P/K)$ states and consists solely of θ and ϕ states. Moreover, $s \in \chi$ is in S^i if and only if there exists a state $t \in \bar{S}^i$ such that $t \rightarrow^* s$, i.e., there is an ε-path to s from an atomic state in S^i. But then by Corollary 4, $s \in S^i \cap \chi$ if and only if there is an ε-path to s from a state in $\bar{S}^i$ with at most one back edge. As in the algorithm of [MyM88], the algorithm of Figure 4 finds the ε-labeled states in S^i by processing the states in χ twice in a topological order relative to F^*'s dag edges. Recall that the subgraph of F^* restricted to dag edges is acyclic and so a topological ordering exists. Such a topological ordering of χ is precomputed in $O(P)$ time† and used by each invocation of $\varepsilon - closure()$ to process the states once in this order.

† To produce a topological listing of χ, it suffices to traverse T_R top-down and left-to-right. When module V's root is first visited append θ_V to the list, and when leaving V's root append ϕ_V to the list.

At the beginning of the phase, *Set* models $\bar{S}^i$, and as the phase proceeds, states in χ are added to the state-set modeled by *Set*. When an ε-labeled state s is processed, its bit is set in the integer $Set[Mod(s)]$ if and only if $Reach[s, Set[Pre(s)]]$ is true. Every predecessor of s is in $\Pi_{Pre(s)}$. Thus s's bit is set if and only if it is a successor of a state in the state-set currently modeled by *Set*. It follows by induction that *Set* models a subset of S^i and a superset of $\bar{S}^i$ at all times during the phase. It remains to prove that at the end of the phase, every state s in $S^i \cap \chi$ has been added to *Set*. By Corollary 4 there is an ε-path $p = t \rightarrow \cdots \rightarrow r \rightarrow s$ such that it contains zero or one back edges and $t \in \bar{S}^i$. If $r \in \bar{S}^i$ then certainly s is added to *Set*. Here after, assume $r \in \chi$ or, equivalently, that $r \neq t$. If p contains no back edges then we claim by induction that s is added when it is processed in the first call to $\varepsilon - closure()$. Suppose it is true for all predecessors of s in the topological order. r is processed before s since $r \rightarrow s$ is a dag edge, and r is also reached from t along a path with no back edges. Thus, by the induction hypothesis, r has been added before s is processed. But then s will be added since $r \rightarrow s$. Finally, we claim that if p contains one back edge then s is added when it is processed in the second call to $\varepsilon - closure()$. Suppose it is true for all predecessors of s in the topological order. If $r \rightarrow s$ is a dag edge, then r is processed before s in the second call and r satisfies the induction hypothesis. Thus, in this case, r has been added before s and so s is also added. Otherwise $r \rightarrow s$ is a back edge, and r is reached from t along a path with no back edges. Thus, r is added in the first call to $\varepsilon - closure()$ and s is added in the second call. For those familiar with the data flow analysis literature, the correctness of the closure phase follows simply from the fact that the list obtained by concatenating two copies of the topological order of states in χ is a *node listing* [Ken75] for the set of all ε-paths to states in χ with at most one back edge.

The algorithm of Figure 4 starts by initializing *Set* to model the state-set $\{ \theta \}$ in lines 1-3. After executing $\varepsilon - closure()$ in line 4, *Set* models the set $\{ s : \theta \rightarrow^* s \}$ as there must be a path with no dag edges to s by Corollary 4. But $\{ s : \theta \rightarrow^* s \} = S^0$ and so lines 1-4 correctly start the state-set simulation. Upon completion of the main loop of lines 5-9, *Set* models the set $S^n - \{\phi\}$. Thus in line 10, $Reach[\phi, Set[Pre(\phi)]]$ is true if and only if ϕ is the successor of a state in S^n, which in turn is true if and only if A matches a word in the language denoted by F^*. Thus the algorithm reports a match exactly when there is one.

Each call to ε-closure takes $O(P/K)$ time as do the loops in lines 1-2 and 6-7. It then follows that the algorithm takes a total of $O(PN/K)$ time. Construction of the tables takes $O(P2^K)$ time and they occupy $O(P2^K/K)$ space. Combining the preprocessing and scanning steps, we have a $O(P(N/K + 2^K))$ time, $O(P2^K/K)$ space algorithm for a $(K+1)$-bit uniform RAM. As long as $K < lg\ (N/K)$, the algorithm requires $O(PN/K)$ time and $O(PN/K^2)$ space. Thus on a lgN-bit uniform RAM, we can choose K to be $(lg\ N)/2$ and the algorithm requires $O(PN/lg\ N)$ time and $O(PN^{1/2}/lg^2\ N)$ space.

```
1.  for V a module do
2.      Set[V] ← 0
3.  Set[Mod(θ)] ← Setbit[Bit(θ), Set[Mod(θ)]]
4.  ε-closure()
5.  for i ← 1 to N do
      { # Module Phase: Set[V] = INT(Π_V ∩ S^{i-1}) for all V. #
6.      for V a module do
7.          Set[V] ← Goto[V, Set[V], a_i]
        # Closure Phase: Set[V] = INT(Π_V ∩ S^i ∩ L_{a_i}) for all V. #
8.      ε-closure()
9.      ε-closure()
      }
    # Set[V] = INT(Π_V ∩ S^n) for all V. #
10. if Reach[φ, Set[Pre(φ)]] then
11.     ''Match A''

    procedure ε-closure()
12.   { for s ∈ χ in topological order do
13.       if Reach[s, Set[Pre(s)]] then
14.           Set[Mod(s)] ← Setbit[Bit(s), Set[Mod(s)]]
      }
```

FIG. 4. The $O(PN/K)$ state-set simulation of F^* on A.

4.3 INFINITE ALPHABETS. We now consider the case where Σ is a countably infinite set, e.g., the set of all words over some finite alphabet. Assume that a total ordering of Σ exists, and that two symbols may be compared in $O(C)$ time. The key observation is that Σ_R, the set of symbols *used* in R, contains no more than P symbols and hence is finite.

Using an optimal comparison-based sort, a sorted list L_R of the symbols in Σ_R is built in $O(CPlg\ P)$ time. Let $Code(a)$ be symbol a's position in L_R if $a \in \Sigma_R$, and 0 otherwise. $Code(a)$ can be computed in $O(Clg\ P)$ time using a binary search over the list. For each module V, let Σ_V be the set of symbols labeling the atomic leaves of V. As above, a map $Code_V(a)$ is realized by producing a sorted list of Σ_V for each V in a total of $O(CPlg\ K)$ time. In $O(CPlg\ P)$ additional time, $Code(\lambda(s))$ and $Code_V(\lambda(s))$ is computed and stored with each state $s \in \Pi_V$ so that these quantities need not be recomputed during the construction of the following altered tables.

Noting that $Goto[V, Z, a] = 0$ if $a \notin \Sigma_V$, we reduce the range of the third index from Σ to the integers in the interval $[1, |\Sigma_V|]$. That is, for each $a \in \Sigma_V$, $Goto[V, Z, Code_V(a)] \equiv Goto[V, Z, a]$. With this change, $O(K2^K)$ time and space is needed to compute the *Goto* table of a module. In addition, for each $a \in \Sigma_R$ the scanning step needs a list $Cont[Code(a)]$ of the modules containing an a-labeled state and the symbol's code in that module.

Formally, $Cont[Code(a)] = \{ (V, Code_V(a)) : a \in \Sigma_V \}$ and this table of lists can be built in $O(P)$ time with a bucketing approach.

With these new tables, the module-phase of the scanning step consists of computing $c = Code(a_i)$ for the current symbol a_i, and then looking up the *Goto* transition for each V in $Cont[c]$. For V not on $Cont(c)$, the state set of V becomes the empty set. Thus we can replace lines 6 and 7 of Figure 4 with:

```
for V a module do
    Temp[V] ← 0
c ← Code(a_i)
if c ≠ 0 then
    for (V, d) ∈ Cont[c] do
        Temp[V] ← Goto[V, Set[V], d]
for V a module do
    Set[V] ← Temp[V]
```

This code fragment takes $O(P/K + Clg\ P)$ time, for a total over the entire scanning step of $O(PN/K + CNlg\ P)$ time. Table construction still takes $O(P2^K)$ time but now also takes this much space. There is also the smaller $O(CPlg\ P)$ term for building sorted translation lists. In summary, infinite alphabets can be accommodated at an additional cost of $O(CNlg\ P)$ time.

5. *Experience with a Practical Algorithm for Bit Vector Machines*

We now assume more realistically that a K-bit machine can perform bit-wise logical operations | (or) and & (and). With these operations we can express the quantities in the tables *Setbit*, *Reach*, and *Goto* with $O(1)$-computable expressions involving four smaller tables: *Power*, *Pred*, *Succ*, and *Atom*. The list below defines the four tables and how they specify the original three.

(1) $Setbit[Z, k] \equiv Z \mid Power[k]$

where $Power[k] = 2^k$.

(2) $Reach[s, Z] \equiv (Z\ \&\ Pred[s] \neq 0)$

where $Pred[s] = BIT(\{ t : t \rightarrow s \})$.

(3) $Goto[V, Z, a] \equiv Succ[V, Z]\ \&\ Atom[V, a]$

where $Succ[V, INT(X)] = INT(\{ s$: There exists $t \in X$ s.t. $t \rightarrow s$ and s is atomic $\})$

and $Atom[V, a] = BIT(\{ s : s \in \Pi_V$ and $\lambda(s) = a \})$.

Power consists of K integers and takes $O(K)$ time to compute by successive doubling. The *Pred* table consists of $2M$ integers, one for each ε-labeled state. Each integer takes $O(K)$ time to compute for a total of $O(P)$ time for the

entire table. For each module V, its *Succ* table consists of $2^{|\Pi_v|}$ integers that can be computed in $O(2^K)$ time with a recurrence like that for *Goto* and *Reach*. V's *Atom* table consists of $|\Sigma|$ integers and is easily computed in $O(|\Sigma|+K)$ time. Thus the time to compute the four new tables is reduced to $O(P2^K/K)$ time overall. Moreover, the total space required is less than $[\frac{2^{K+1}}{K-1} + \frac{3|\Sigma|}{K} + O(\frac{1}{K})]\ L$ integers in the worst case. The first term is for the *Succ* tables, the second for the *Atom* tables, and the remainder is an insignificant term in practice. Recall that L is the number of leaves in the parse tree for the regular expression. For example, over the ASCII alphabet at most $270L$ integers are needed for $K = 10$, and $2520L$ for $K = 14$.

Software of the same functionality as UNIX *egrep* [Aho80] was built using our bit-vector algorithm for regular expression pattern matching. A number of pragmatic optimizations were employed. As in *egrep*, character classes are implemented as a single atomic state. Second, the tables are built directly from T_R; F^* is never explicitly constructed. This is done by computing the predecessor and successor relations of states in F^* directly from T_R in two $O(P)$ sweeps of the tree as observed in Section 3.9 of [ASU85]. Third, *Atom* and *Succ* tables are *not* built for modules all of whose leaves are modular. For such modules V, *Set*$[V]$ is set to zero as opposed to *Succ*$[V, Z]$ *&* *Atom*$[V, a] = 0$ for all symbols a. Fourth, only those vertices in χ that are within the ''scope'' of a back edge are traversed in the second call to $\varepsilon - closure()$ in line 9 of Figure 4. A vertex v needs to be processed a second time only if it can follow a back edge in a non-cyclic path and is in a different module than the one containing the back edge. This is true if and only if there is a * or + operator on the path from T_R's root to the root of v's module. Note that when the regular expression does not contain * or + operators, this optimization halves the time spent in $\varepsilon - closure()$ by removing the entire second sweep.

Our final optimization takes advantage of the particularly simple structure afforded our node-listing approach. Once R has been processed and before A is scanned, so much is known about the control flow and values in the inner loops of lines 6 to 9 that significant time gains can be achieved by compiling code for these lines that is tailored to R [Pen86, Mil88]. Figure 5 shows a hypothetical *C*-program tailored to the pattern in Figures 2 and 3. The calls to $\varepsilon - closure()$ are in-line, and each loop in lines 6-7 and 12-14 has been completely unrolled. For each module, *Set*$[V]$ is mapped to a register variable, e.g., *S00* through *S03* in Figure 5. *Succ* and *Atom* tables are also reduced to one-dimensional arrays by naming the table for each module, e.g. *Succ01* and *Atom02*. Also, all occurrences of *Pred*$[s]$ are known and compiled as constants. The values of *Set* computed in lines 1 to 4 are known and so are compiled into initialization values for the S variables. The outer loop of line 5 is realized as a series of buffered reads from the file *Afile*. Further note that the second call to $\varepsilon - closure()$ is back-edge optimized, and *Succ00* is omitted since its module has no atomic leaves. The declarations and initialization values of the *Atom* tables are omitted for brevity.

While the compiled scanner is fast, the time taken to compile it is unacceptable. To circumvent this, we produce object code for the host machine in memory and then call it directly. We performed this optimization on a VAX 8650 running 4.3bsd UNIX and obtained the following timing estimates. The time to build the tables and scanner object code is less than $M(2^{K+1} \cdot 2.0\mu s + 1ms)$. The time taken to construct each module is thus about $65ms$ when $K = 14$, and about $5ms$ when $K = 10$. The scanner processes characters at a rate of one every $M \cdot .46\mu s + X \cdot .35\mu s + .44\mu s$ where X is the number of vertices processed in the two calls to $\varepsilon - closure()$, e.g., $X = 8$ in Figure 5. X varies between $2(M-1)$ and $4(M-1)$ depending on the efficacy of the back-edge optimization. Thus for patterns that require one module, the scanner runs at $.9\mu s$ per character; for two modules between 2.06 and 2.76 μs; for three between 3.22 and 4.62; and so on. The variance in the time spent per character is very small, since there is very little conditional execution in the scanner's inner loop.

```
#include <stdio.h>

succ01[] = { 000, 006, 006, 006, 010, 016, 016, 016,
             006, 006, 006, 006, 016, 016, 016, 016,
           };

succ02[] = { 000, 002, 004, 006, 000, 002, 004, 006,
             002, 002, 006, 006, 002, 002, 006, 006,
           };

succ03[] = { 000, 006, 000, 006, 000, 006, 000, 006, };

main()
{   int ifile, num;
    char buf[BUFSIZ+1];
    register char *s;
    register int   a;
    register int S00=07, S01=01, S02=01, S03=0;
    ifile = open("Afile",0);                          /* Scan loop */
    while ((num = read(ifile,buf,1024)) > 0)
    {   buf[num] = 0;
        for (s = buf; a = *s; s++)
        {   S00 = 0;                                  /* Module phase */
            S01 = succ01[S01] & atom01[a];
            S02 = succ02[S02] & atom02[a];
            S03 = succ03[S03] & atom03[a];
            if (S00 & 001) S01 |= 001;                /* ε-closure() */
            if (S01 & 013) S00 |= 002;
            if (S00 & 001) S02 |= 001;
            if (S02 & 004) S03 |= 001;
            if (S03 & 006) S02 |= 010;
            if (S02 & 011) S00 |= 004;
            if (S02 & 004) S03 |= 001;                /* ε-closure() */
            if (S03 & 006) S02 |= 010;
        }
    }
    exit (S00 & 00006);
}
```

FIG. 5. Compiled scanner for F^* of Figure 3.

We compared the scanning speeds over ASCII texts of our program, called *mgrep*, against an implementation of the standard state-set simulation, called *sgrep*, and another employing Aho's cache-based deterministic simulation used in the UNIX System 5 implementation of *egrep* [Aho80, ASU(§3.7)]. Care was taken to fully optimize these implementations too, as evidenced by the fact that our implementation of *egrep* is faster than the *UNIX* implementation. Table 1 shows the scanning speed of the three programs on a set of patterns chosen to illustrate the following points. For all patterns, *mgrep* with $K = 14$ uses one or two modules and performs exactly as characterized in the paragraph above. Note that when $L \leq K$, *mgrep* effectively builds a single DFA for the pattern and so, not surprisingly, is very fast. *egrep*'s $O(1)$ expected-time algorithm scans a character every $1.6\mu s$ as long as its cache does not overflow frequently. This happens only when the pattern's deterministic finite automaton has a large number of states and many of them are reached during the scan, e.g., $p.... f = p.^4 f$. Such patterns are quite infrequent, but when they are used, *egrep*'s performance becomes worse than *sgrep*'s as shown in the last two patterns of Table 1. The expected time complexity of the standard state-set simulation is significantly better than its $O(PN)$ worst-case complexity suggests. *sgrep* consumes between $3S\mu s$ and $7S\mu s$ per character scanned where S is the average number of states visited when advancing its state set. The size of S is generally much less than P and it depends on the degree of alternation in the vicinity of the start state. The size of S for several patterns is shown in Table 1.

PATTERN	*mgrep (M,X)*	*egrep*	*sgrep (S)*
$printf$	0.9 (1,0)	1.6	4.5 (1.06)
$printf \mid while \mid else$	2.1 (2,2)	1.6	11.5 (4.11)
$(printf \mid while \mid else)^*$	2.7 (2,4)	1.6	19.0 (6.11)
$[a-z][a-z0-9]^*$	0.9 (1,0)	1.6	14.5 (2.30)
$p.^{20} f$	2.1 (2,2)	$12.3^\dagger$	7.0 (1.57)
$[a-z].^6 f$	0.9 (1,0)	$21.4^\dagger$	19.8 (3.43)

† - Time is 1.6 for $p.^{15} f$ and $[a-z].^5 f$.

TABLE 1. Scanning speed of three algorithms in μs per character.

In practice it is the expected time complexity of the underlying algorithm that determines a tool's empirical performance. Thus while our algorithm represents a worst-case improvement over the *sgrep* algorithm, *mgrep*'s performance in practice demonstrates that it is the most effective for small patterns. This is because *mgrep*'s expected scanning speed is $\Theta(P/K)$ while that of *egrep* is usually $\Theta(1)$ and that of *sgrep* is $\Theta(S)$ where S and P/K are incomparable. As a consequence, *mgrep* is guaranteed to be superior to *egrep* only for patterns with less than 14 leaves, and *sgrep* only for patterns with less than about 40 leaves. However, *mgrep*'s performance is independent of the structure of the pattern and so does not exhibit the explosive behavior of *egrep* on some types of patterns. This fact also implies that *mgrep* is frequently faster than *sgrep* on patterns with more than 40 leaves.

6. *An Open Problem*

The ‘‘Four Russians’’ paradigm has provided algorithms for sequence comparison and regular expression pattern matching that are superior in the worst case. The approximate regular expression matching problem, also known as regular order correction [WaS78], involves the fusion of sequence comparison and exact pattern matching concepts. For this problem, the node-listing algorithm of [MyM88] has a structure that may lend itself to modularization as in this paper. Can the ‘‘Four Russians’’ paradigm be successfully applied to approximate regular expression pattern matching?

ACKNOWLEDGEMENTS. The author wishes to thank the referees for their effort and comments that lead to an improved paper.

REFERENCES

[Aho80] Aho, A.V. ‘‘Pattern matching in strings.’’ *Formal Language Theory* (R. Book, ed.), Academic Press (1980).

[AHU75] Aho, A.V., J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1975).

[AKD70] 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), 487-488 (in Russian). English translation in *Soviet Math. Dokl.* **11** (1975), 1209-1210.

[ASU85] Aho, A.V., R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley (1985).

[Gal85] Galil, Z. ‘‘Open problems in stringology.’’ *Combinatorial Algorithms on Words* (A. Apostolico and Z. Galil, eds.), Springer-Verlag (1985), 1-8.

[HeU75] Hecht, M.S. and J.D. Ullman. ‘‘A simple algorithm for global flow analysis programs.’’ *SIAM J. Computing* **4**, 4 (1975), 519-532.

[Ken75] Kennedy, K. ‘‘Node listing techniques applied to data flow analysis.’’ *Proc. 2nd ACM Conf. on Principles of Programming Languages* (1975), 10-21.

[MaP80] Masek, W.J. and M.S. Paterson. ‘‘A faster algorithm for computing string-edit distances.’’ *J. Computer and System Science* **20**, 1 (1980), 18-31.

[MaP83] Masek, W.J. and M.S. Paterson. ‘‘How to compute string-edit distances quickly.’’ *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, D. Sankoff and J.B. Kruskal, eds., Addison-Wesley (1983), 337-349.

[Mil88] Miller, W. ‘‘Efficient searching of biosequence databases.’’ Tech. Rep. CS-88-34, Dept. of Computer Science, The Pennsylvania State University, University Park, PA 16802.

[MyM88] Myers, E.W., and W. Miller. ‘‘Approximate matching of regular expressions.’’ *Bull. Math. Biol.* **51**, 1 (1988), to appear.

[Pen86] Penello, T.J. ‘‘Very fast LR parsing.’’ *ACM SIGPLAN Notices* **21**, 7 (1986), 145-150.

[Sed83] Sedgewick, R. *Algorithms*, Addison-Wesley (1983).

[Tho68] Thompson, K. ‘‘Regular expression search algorithm.’’ *Comm. ACM* **11**, 6 (1968), 419-422.

[WaF74] Wagner, R.A. and M.J. Fischer. ‘‘The string-to-string correction problem.’’ *J. ACM* **21** (1974), 168-173.

[WaS78] Wagner, R.A. and J.I. Seiferas. ‘‘Correcting counter-automaton-recognizable languages.’’ *SIAM J. Computing* **7**, 3 (1978), 357-375.