Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning

Monika Rauch Henzinger
Systems Research Center, Digital Equipment Corporation, Palo Alto, CA *

Jan Arne Telle
Department of Informatics, University of Bergen, Norway †

Abstract

This paper shows how a general technique, called *lock-step search*, used in dynamic graph algorithms, can be used to improve the running time of two problems arising in program verification and communication protocol design.

(1) We consider the *nonemptiness problem for Streett automata*: We are given a directed graph $G = (V, E)$ with $n = |V|$ and $m = |E|$, and a collection of pairs of subsets of vertices, called *Streett pairs*, $\langle L_i, U_i \rangle, i = 1..k$. The question is whether G has a cycle (not necessarily simple) which, for each $1 \leq i \leq k$, if it contains a vertex from L_i then it also contains a vertex of U_i. Let $b = \sum_{i=1..k} |L_i| + |U_i|$. The previously best algorithm takes time $O((m+b) \min\{n, k\})$. We present an algorithm that takes time $O(m \min\{\sqrt{m \log n}, k, n\} + b \min\{\log n, k\})$.

* *monika@src.dec.edu.* This research was done while at the Department of Computer Science, Cornell University, Ithaca, NY and was supported by an NSF CAREER Award, Grant No. CCR-9501712.

† *telle@ii.uib.no.* This research was done while visiting the Department of Computer Science, Cornell University, Ithaca, NY and was supported by a fellowship from the Norwegian Research Council.

(2) In *communication protocol pruning* we are given a directed graph $G = (V, E)$ with l special vertices. The problem is to efficiently maintain the strongly-connected components of the special vertices on a restricted set of edge deletions. Let m_i be the number of edges in the strongly connected component of the ith special vertex. The previously best algorithm repeatedly recomputes the strongly-connected components which leads to a running time of $O(\sum_i m_i^2)$. We present an algorithm with time $O(\sqrt{l}\sum_i m_i^{1.5})$.

1 Introduction

Maintaining the strongly-connected components of a digraph $G = (V, E)$ efficiently under vertex or edge deletions is an unsolved problem. No data structure is known that is faster than recomputation from scratch. This is unfortunate since such a data structure would speed up various algorithms. In this paper we describe two such algorithms and show how a technique, called *lock-step search*, used in dynamic graph algorithms, can improve their running time.

Nonemptiness for Streett automata
A Streett automaton with *Streett pairs* $\langle L_i, U_i \rangle$ is an automaton on infinite words where a run is accepting if for all pairs i, if the run visits infinitely many times some state in L_i then it also visits infinitely many times some state in U_i. The first problem that we consider is called the *nonemptiness problem for Streett automata*: Given a directed graph $G = (V, E)$ and a collection of pairs of subsets of vertices, called *Streett pairs*, $\langle L_i, U_i \rangle, i = 1..k$, determine if G has a cycle (not necessarily simple) which, for each $1 \leq i \leq k$, if it contains a vertex from L_i then it also contains a vertex of U_i. [1]

Nonemptiness checking of Streett automata is used in computer-aided verification. Consider, for example, the problem of checking if a strongly fair finite-state system A satisfies a specification ϕ in linear temporal logic ("model checking"). The system A can be modeled as a Streett automaton. The negated specification $\neg\phi$ can be translated into an equivalent Büchi automaton, and

[1]Note that the nonemptiness problem for Streett automata usually includes also a designated *root* vertex (the start state) and requires that the cycle asked for be reachable from the root. For ease of presentation, we assume that a simple linear preprocessing step has computed the input graph G consisting only of vertices reachable from the root, since the other vertices will not affect the solution.

therefore Streett automaton, $B_{\neg\phi}$. Then model checking reduces to checking the nonemptiness of the product Streett automaton $A \times B_{\neg\phi}$ [7].

Let $|V| = n, |E| = m$ and $b = \sum_{i=1..k} |L_i| + |U_i|$. The previously best algorithms for this problem take time $O((m + b) \min\{n, k\})$ [1, 3]. We present an $O(m \min\{\sqrt{m \log n}, k, n\} + b \min\{\log n, k\})$ algorithm for the problem. The improved running time is achieved through (1) lock-step search and (2) an efficient data structure for representing the Streett pairs $\langle L_i, U_i \rangle$. In model checking, frequently G has bounded out-degree. In this case $m = O(n)$ and our algorithm has running time $O(n \min\{\sqrt{n \log n}, k\} + b \min\{\log n, k\})$.

Protocol Pruning

A communication system defines interactions between different components using exact rules, called *protocols*. Since protocol standards have become very complex, various approaches try to simplify protocols. A new technique by Lee, Netravali, and Sabnani [4] models a protocol as a collection of communicating finite state machines, and prunes the protocol without constructing the composite machine. The finite state machines are represented as a directed graph with l special vertices (start states), one per machine. Interactions between machines are modeled as dependencies among edges. Their algorithm repeatedly "prunes off" (i.e. deletes) edges of the graph and recomputes the strongly-connected components of the special vertices. Which edges are deleted in the next iteration depends on dependencies between the edges left in the current strongly-connected components of the special vertices. If the strongly-connected components have not changed between two iterations the algorithm terminates with these strongly-connected components representing the pruned protocol machine.

Let m_i be the number of edges in the strongly-connected component of the ith special vertex. Recomputing the strongly-connected components from scratch in each iteration leads to a running time of $O(\sum_i m_i^2)$. We present an algorithm with time $O(\sqrt{l} \sum_i m_i^{1.5})$. For constant l our algorithm takes time $O(\sum_i m_i^{1.5})$.

Lock-Step Search

Both improved algorithms use lock-step search. Lock-step search was introduced by Even and Shiloach [2] to maintain the connected components of an *undirected* graph under edge deletions. After the deletion of the edge (u, v), they start a (depth-first) search for the resulting component "in parallel" at u and at v: the algorithm executes one step of the search at u and one step of the search at v and stops whenever one of the searches is completed.

Let C_u and C_v be the new components of u and v. *Case 1:* If $C_u \neq C_v$, then the lock-step search takes time $O(\min(|C_u|, |C_v|)$, where $|C|$ denotes the number of edges in a component. Charging this cost to the edges in the smaller of the two components charges at most $O(\log n)$ to an edge in the graph: an edge is charged only if the number of edges in its component is at least halved.

Case 2: If $C_u = C_v$ the search takes time $O(|C_u|)$. Thus, if this situation arises at almost all edge deletions, the running time for the whole algorithm is $O(m^2)$.

We use a slightly modified lock-step search on directed graphs, described below, to find strongly-connected components. Note that both the problems we consider allow the algorithm to terminate as soon as a strongly-connected component is found that fulfills their condition. Thus, instead of first finding the strongly connected components and then checking for each strongly-connected component if it fulfills the conditions, our new algorithms first guarantee that the conditions are fulfilled in the current graph and then start a lock-step search. If Case 1 holds, the costs are charged as described above and further deletions might be necessary. If Case 2 arises the traversed strongly-connected component fulfills the conditions and the algorithms terminate.

Lock-Step Search on Directed Graphs

Assume we have a strongly-connected digraph from which the set of edges F are deleted. If the resulting graph G is not strongly-connected then lock-step search can be used to find a strongly-connected component containing at most half the edges of G. Let $Heads = \{v : uv \in F\}$ be the vertices which are heads of edges in F and $Tails = \{u : uv \in F\}$ be the vertices which are tails of edges in F. Consider the condensed graph of G where strongly-connected components are condensed into single vertices. This graph is acyclic, every source vertex represents a source component which must contain a vertex in *Heads* and every sink vertex represents a sink component which must contain a vertex of *Tails*, since before deletion of F the condensed graph had only one vertex. Lock-step searches are started on G at each vertex of *Tails* and also on Reverse(G), where all edges are reversed, at each vertex of *Heads*. A search of G rooted at a vertex in a sink component will explore exactly this component. Likewise, a search of Reverse(G) started in a source component of G will explore exactly that component. A search started in any other component of G will eventually enter a sink component (or source component for the reverse searches), and thus terminate later than the search started in the sink (or source) component itself. By searching from all vertices of *Heads* and *Tails* simultaneously we are thus guaranteed to terminate as soon as the smallest source or sink component is discovered.

Notation

For a graph $G = (V, E)$ we denote the induced graph on $S \subseteq V$ by $G[S]$. We consider only directed graphs and may refer to a strongly connected component simply as a component. We denote the standard linear-time strongly-connected component algorithm [6] by SCC. We describe the algorithm for the Streett automata in Section 2. The protocol pruning algorithm is presented in Section 3.

2 Nonemptiness of Streett automata

The best known algorithm for checking nonemptiness of Streett automata dynamically modifies the input graph by deletion of *bad* vertices. A vertex is bad if it belongs to L_i, for some i, but there is no cycle containing both it and a vertex of U_i. Note that if a run reaches a bad vertex (bad state) then it will not be an accepting run. All other vertices are *good*, but note that they can become bad after some vertex deletions. A strongly-connected component of the current graph is said to be *good* if it contains good vertices only. The goal is then to find a non-trivial good component, or to decide that after iteratively deleting all bad vertices no non-trivial component is left. The algorithm computes the strongly-connected components of the graph, halts if a good component is found or if only trivial components remain, otherwise deletes at least one bad vertex and repeats. A straightforward implementation, using the linear-time depth-first strongly-connected component algorithm SCC [6], gives a running time of $O((m+b)\min\{n,k\})$ [1, 3], for $|V| = n$, $|E| = m$ and $b = \sum_{i=1..k} |L_i| + |U_i|$. We present an $O(m \min\{\sqrt{m \log n}, k, n\} + b \min\{\log n, k\})$ algorithm for the problem.

2.1 Data structure for Streett pairs

The algorithm maintains a partition of the vertex set. Each set in the partition induces a collection of strongly-connected components of the current graph. Note that for a set S of the partition, any vertices in $S \cap L_i$ are bad if $S \cap U_i = \emptyset$. For $S \subseteq V$, define $bits(S) = \sum_{i=1..k} |S \cap L_i| + |S \cap U_i|$. We need to quickly refine a set in the partition and locate and delete bad vertices. For this we use a data structure with operations:

Construct(S) returns the initialized data structure $C(S)$, for $S \subseteq V$.

Remove$(S, C(S), B)$ removes B from S and returns the data structure $C(S \setminus B)$, for $S, B \subseteq V$.

Bad$(C(S))$ returns $\{v \in S : \exists i : v \in L_i \wedge U_i \cap S = \emptyset\}$, for $S \subseteq V$.

Each operation need only return a pointer to the appropriate item. In section 2.3 we show the following result.

Lemma 2.1 *After a one-time initialization of $O(k)$ the data structure can be implemented with running time $O(bits(S) + |S|)$ for Construct(S), $O(bits(B) + |B|)$ for Remove(S, $C(S)$, B) and $O(1)$ for Bad($C(S)$).*

2.2 The Algorithm for Non-Emptiness Testing of Streett Automata

The algorithm maintains a list L of data structures $C(S)$, where $S \subseteq V$ is a vertex subset whose induced subgraph $G[S]$ contains candidates for good components. The vertex subsets on this list are disjoint and the list is initialized to contain the components of the input graph. Bad vertices are removed until only good vertices remain and only then are new components computed. Note that the statement "$C(S) :=$ *Remove*$(S, C(S), B)$" also updates S to $S \setminus B$. Some vertices may be labelled h (for head) or t (for tail) signifying that further exploration rooted at the vertex is necessary. The algorithm halts as soon as a good component is found. The algorithm is given on the next page. We argue separately for correctness and running time for the cases $\min\{n, k\} < \sqrt{m \log n}$ and $\min\{n, k\} \geq \sqrt{m \log n}$.

Emptiness-Testing Algorithm for Streett Automata
Input: A digraph $G = (V, E)$ and Streett pairs $\langle L_i, U_i \rangle, i = 1..k$
Output: A good component, if it exists

1. Run SCC to find components $G[S_1], ..., G[S_d]$ of G.
2. For i=1 to d add $Construct(S_i)$ to list L.
3. **While** $L \neq \emptyset$ **do**
   (a) Get $C(S)$ from L.
   (b) **While** $Bad(C(S)) \neq \emptyset$ **do** $C(S) := Remove(S, C(S), Bad(C(S)))$.
   (c) Add label h (t) to vertices in S which have an incoming (outgoing) edge to a vertex just removed. Let $Heads := \{v \in S : h \in label(v)\}$ and $Tails := \{v \in S : t \in label(v)\}$.
   (d) Case 1. **If** $|Heads| + |Tails| = 0$ **do** /* $G[S]$ is a good component */
      i. If $|S| \geq 2$ then HALT and output $G[S]$
   (e) Case 2. **Else if** $|Heads| + |Tails| \geq \sqrt{m/\log n}$ **do** /* If $\min\{n, k\} < \sqrt{m \log n}$) change this line to: "Case 2. Else do ".*/
      i. Remove all h, t-labels from S
      ii. Run SCC to find comp.'s $G[S_1], ..., G[S_d]$ $(|S_1| \geq |S_i|)$ of $G[S]$.
      iii. For i=2 to d
         $C(S) := Remove(S, C(S), S_i)$
         Add $Construct(S_i)$ to L
      iv. Add $C(S)$ to L /* Note that $S = S_1$ */
   (f) Case 3. **Else** $0 < |Heads| + |Tails| < \sqrt{m/\log n}$ **do** /* If $\min\{n, k\} < \sqrt{m \log n}$) remove this case completely */
      i. For each $v \in Tails$ search $G[S]$ from v and for each $v \in Heads$ search Reverse$G[S]$ (all edges reversed) from v. Run the $|Tails| + |Heads|$ searches in lock-step until the first search terminates with all vertices R reachable from its root visited.
      ii. If $R = S$ then HALT and output $G[S]$.
      iii. Remove all h, t-labels from R. Add label h (t) to vertices in $S \setminus R$ which have an incoming (outgoing) edge to R.
      iv. Add $Remove(S, C(S), R)$ to L
      v. Add $Construct(R)$ to L
4. L empty, HALT and output: "No good components exist".

Lemma 2.2 *The algorithm for nonemptiness testing of Streett automata is correct when* $\min\{n, k\} < \sqrt{m \log n}$.

Proof. Line 3(e) is then: "Case 2. Else do" and Case 3 does not exist. The while loop in line 3 has the invariant: "For any $C(S)$ in the list L the vertices S have no h, t-labels, $G[S]$ is a component of the current graph and all candidates for non-trivial good components are in L". The invariant is maintained since after removal of bad vertices from S all h, t-labels are removed from S and components of $G[S]$, the new candidates, are added to L. If $Bad(C(S)) \neq \emptyset$ then after removal of bad vertices some vertex in the remainder (unless remainder is empty) has an edge to a removed vertex, since we started with a component. Thus, if $|Heads| + |Tails| = 0$ and S contains at least two vertices then $G[S]$ is a non-trivial component with no bad vertices. We conclude that the algorithm finds a good component, if it exists. □

Lemma 2.3 *The algorithm for nonemptiness testing of Streett automata has running time* $O(m \min\{k, n\} + b \min\{\log n, k\})$ *when* $\min\{n, k\} < \sqrt{m \log n}$.

Proof. Each time an edge uv is involved in a call of SCC in line 3(e)ii, the component $S(uv)$ the edge belonged to had some bad vertices removed from it, in line 3(b). Since each such removal decreases both $|\{i \in \{1..k\} : L_i \cap S(uv) \neq \emptyset\}|$ and $|\{w \in V : w \in S(uv)\}|$ by at least one, the total cost of calls to SCC is bounded by $O(m \min\{k, n\})$. Deletions of bad vertices in line 3(b) take total time $O(b) = \sum_{v \in V} bits(v)$ since a vertex v is deleted at most once at a cost of $O(bits(v))$. Note that the partitioning of the data structure for $C(S)$ into $C(S_1), ..., C(S_d)$ in the for loop of line 3(e)iii avoids the cost of $bits(S_1)$ for the largest component S_1. After removing $S_i, i = 2..k$ from $C(S)$ then precisely $C(S_1)$ remains. The cost of Remove$(S, C(S), S_i)$ and Construct(S_i) of $O(bits(S_i) + |S_i|)$ is charged to vertices of S_i or to their bit in L_j or U_j, $2 \leq i \leq d$. When a vertex v is charged, the size of its component $S(v)$ is therefore halved, and $|\{i \in \{1..k\} : L_i \cap S(v) \neq \emptyset\}|$ decreases. The total cost of partitioning is then $O((b + n) \min\{\log n, k\})$, and the total running time is $O(m \min\{k, n\} + b \min\{\log n, k\})$ for this case. □

Lemma 2.4 *The algorithm for nonemptiness testing of Streett automata is correct when* $\min\{n,k\} \geq \sqrt{m \log n}$.

Proof. We explain the parallel searches in line 3(f)i. Assume that $G[S \cup B]$ is a component of the current graph from which we delete the vertices B. Let *Heads* (*Tails*) be the vertices in S with an incoming (outgoing) edge to a vertex of B. Consider the condensed graph of $G[S]$ where strong components are condensed into single vertices. This graph is acyclic, every source component must contain a vertex in *Heads* and every sink component must contain a vertex of *Tails*, since $G[S \cup B]$ was strongly-connected. A search of $G[S]$ rooted at a vertex in a sink component will explore exactly this component, and likewise a search of Reverse$G[S]$ started in a source component will explore exactly that component. A search started in any other component will eventually enter a sink component (or source component for the reverse searches), and thus terminate later than the search started in the sink (or source) component itself. By searching from all vertices of *Heads* and *Tails* simultaneously we are thus guaranteed to terminate as soon as the smallest source or sink component is discovered. We can now state the invariant of the while loop: "For any $C(S)$ in the list L, either $G[S]$ is a component or any source (sink) component of $G[S]$ has a vertex labelled h (t). All candidates for non-trivial good components are in L". The algorithm takes a data structure from L, deletes bad vertices until only good vertices remain and adds labels h (or t) to vertices which had an incoming (or outgoing) edge removed. The invariant is maintained by Case 2 since it adds unlabelled components to the list. Case 3 searches all vertices labelled h and t simultaneously. As described above it returns the smallest source or sink component R of $G[S]$. If $R = S$ then we have a good component. Otherwise, we add the unlabelled component R to L and from S we delete vertices R and add labels h (or t) to vertices which had an incoming (or outgoing) edge to R, before adding $S \setminus R$ to L. Thus the loop invariant is maintained and the algorithm is correct. $\square$

Lemma 2.5 *The algorithm for nonemptiness testing of Streett automata has running time* $O(m\sqrt{m \log n} + b \min\{\log n, k\})$ *when* $\min\{n,k\} \geq \sqrt{m \log n}$.

Proof. The cost of all searches in Case 3 are charged to the edges of the smallest component $G[R]$. If $R = S$ then we halt and each edge is charged $O(\sqrt{m/\log n})$ units. Otherwise, the size of R is at most half the size of S. Thus, each edge is charged at most $\log n$ times. Each time it is charged $O(\sqrt{m/\log n})$ constant units, for total cost of searches in Case 3 of $O(m\sqrt{m \log n})$. Line 3(e) Case 2 occurs after deletion of at least $\sqrt{m/\log n}$ edges from the graph, which can happen at most $\sqrt{m \log n}$ times, each time invoking SCC for a total cost of $O(m\sqrt{m \log n})$. The earlier analysis of data structure operations not under Case 3 still holds. The cost of $O(bits(R) + |R|)$ for the data structure operations in Case 3 are charged to R, and as before each vertex v is charged at most $O((bits(v) + 1) \min\{\log n, k\})$ units total. The total running time for this case is therefore $O(m\sqrt{m \log n} + b \min\{\log n, k\})$. $\square$

We have shown the following theorem.

Theorem 1 *Given a directed graph on n vertices and m edges, and Streett pairs $\langle L_i, U_i \rangle, i = 1..k$, with $b = \sum_{i=1..k} |L_i| + |U_i|$, there is an $O(m \min\{\sqrt{m \log n}, k, n\} + b \min\{\log n, k\})$ time algorithm which either finds a good non-trivial component of the graph or decides that no such component exists, thereby solving the nonemptiness problem for Streett automata.*

2.3 Implementation of data structure for Streett pairs

We prove Lemma 2.1. Assume the Streett pairs are given by sets of circular linked lists, one for each vertex. The list of $v_i \in V = \{v_1, v_2, ..., v_n\}$ contains an entry for all sets L_j such that $v_i \in L_j$ and all sets U_j such that $v_i \in U_j$. The entry specifying that $v_i \in L_j$ (or $v_i \in U_j$) is called the *m-bit* (membership-bit) $[v_i, L_j]$ (or $[v_i, U_j]$) and specifies the data v_i and L_j (or U_j). There are thus a total of b m-bits, each one belonging to the unique *vertex list* specified by the vertex in the m-bit. The data structure we describe constructs analogous *set lists* so that each m-bit also belongs to a unique set list specified by the Streett set in the m-bit. For a given vertex set $S \subseteq V$ we maintain for each L_i such that $L_i \cap S \neq \emptyset$ and for each U_j such that $U_j \cap S \neq \emptyset$ a set list. The set list of $L_i \cap S$ is a doubly linked list of all m-bits $[v, L_i]$ where $v \in S$, in arbitrary order, and the set list for $U_i \cap S$ is defined analogously. The first element of the set list $L_i \cap S$ (or $U_i \cap S$) is called $first(L_i, S)$ (or $first(U_i, S)$). The data structure $C(S)$ for $S \subseteq V$ consists of

- a doubly linked list of vertices in S. Note that each vertex has a pointer to its vertex list.
- the doubly linked set lists $L_i \cap S$ and $U_i \cap S$.
- a doubly linked list of records $\langle first(L_i, S), first(U_i, S)\rangle$, for each i such that $L_i \cap S \neq \emptyset$, and a pointer from each $first(L_i, S)$ and $first(U_i, S)$ to the corresponding record. A record is bad if $L_i \cap S \neq \emptyset$ and $U_i \cap S = \emptyset$.
- a doubly linked list of bad records.

Bad($C(S)$**)** in $O(1)$ time returns a pointer to the list of bad records.

Construct(S**)** is given the list of vertices in S and needs to construct $C(S)$. Every Construct operation uses the same auxiliary array $A[1..k]$, which is initialized only once at the beginning. After each use A is cleaned up in time independent of k, by keeping track of accessed indices. While constructing $C(S)$, the entry $A[i]$ stores the record $\langle first(L_i, S), first(U_i, S)\rangle$, initially $\langle nil, nil\rangle$. To construct $C(S)$, traverse the vertex list for each $v_j \in S$ and add each encountered m-bit to its corresponding set list, e.g. $[v_j, L_i]$ is added to the list with first element $first(L_i, S)$. A final traversal of the accessed indices of array A sets up the lists of records for non-empty set lists and the list of bad records. Apart from the one-time $O(k)$ initialization of the auxiliary array, the Construct(S) operation takes time $O(bits(S) + |S|)$, since there are $bits(S)$ m-bits in vertex lists of S.

Remove($S, C(S), B$**)** is given S, the data structure $C(S)$ and a list of vertices in B. To construct $C(S \setminus B)$ traverse the vertex list for each vertex in B and remove each encountered m-bit from its corresponding set list (note that set lists are doubly linked). If the last vertex of $S \cap U_i$ is removed then add the record $\langle first(L_i, S), first(U_i, S)\rangle$ to the list of bad records. If the last vertex of $S \cap L_i$ is removed then remove the record $\langle first(L_i, S), first(U_i, S)\rangle$ from the list of records and, if present, from the list of bad records. Finally, remove vertices in B from the (doubly linked) vertex list of S. The Remove($S, C(S), B$) operation takes time $O(bits(B) + |B|)$.

3 Protocol Pruning

A new technique called protocol pruning is given in [4] to simplify communication protocols. The input to this problem consists of

1. a labeled digraph $G = (V, E)$ with $m = |E|$ and $n = |V|$ and a labeling function *label* : $E \rightarrow \{1, ..., m\}$;

2. l vertices $a_1, \ldots, a_l$ of V marked as *special* vertices, such that each special vertex belongs to a different strongly-connected component of G;

3. a function *depends_on* : $\{1, ..., m\} \rightarrow \{1, ..., m\}$;

4. an initial set D of edges that have to be deleted from G.

We call an edge e *bad* if the current graph does not contain an edge with label *depends_on*(*label*(e)). An edge e is called *useless* if it is not contained in a strongly-connected component of a special vertex in the current graph. The protocol pruning approach repeatedly removes all edges in D and creates a new set D consisting of bad or useless edges. When the current graph does not contain any bad or useless edges then the components of the special vertices constitute the pruned protocol.

Let m_i be the number of edges in the strongly-connected component of a_i in the initial graph. The previously best implementation of protocol pruning, see [4], adds all bad or useless edges to D. To determine the useless edges it recomputes the strongly connected component of a vertex a_i every time the component lost an edge. Thus, this implementation takes time $O(\sum m_i^2)$.

We give an implementation of protocol pruning in time $O(\sqrt{l} \sum_i m_i^{1.5})$. Our algorithm maintains disjoint sets $S_1, ..., S_l$ of vertices of the current graph G with the special vertex $a_i \in S_i$ so that each induced graph $G[S_i]$ is a collection of strongly-connected components. As before, the algorithm maintains sets *Heads* and *Tails* of vertices. For each vertex v in *Heads* (*Tails*) incoming (outgoing) edges incident to v have been deleted, and the search for the strongly-connected component of v has not yet been completed.

Before each iteration of the main loop we ensure the current graph has no bad edges. The main loop of the algorithm has two cases. If there exists an i with $|S_i \cap \mathit{Heads}| + |S_i \cap \mathit{Tails}| \geq \sqrt{m_i/l}$, then for each such i, the strongly-connected components of $G[S_i]$ are computed from scratch, and D is set to be all newly discovered useless edges. Otherwise, the algorithm starts a search at all vertices in $Heads \cup Tails$. The searches are lock-step in two levels. On the outer level the searches alternate between the different $G[S_i]$, running a fixed number of search steps in each $G[S_i]$. On the inner level all searches within a $G[S_i]$ are also done in lockstep. The search rooted at vertex v tries to determine the strongly-connected component of v. As in the previous algorithm the search for $v \in Tails$ is started in the current graph and the search for $v \in Heads$ in the reverse of the current graph. If all the searches terminate in the exact same component as they were started then there are no useless edges and we are done. Else, assume the first search to terminate is rooted in S_r reaching the vertices $R \subset S_r$. If the special vertex a_r belongs to R then we set D to be all edges of $G[S_r]$ which do not have both endpoints in R and update S_r to R. If a_r does not belong to R then D is set to be all edges with at least one endpoint in R and S_r is updated to $S_r \setminus R$. In this latter case the new $G[S_r]$ is not necessarily strongly connected, so vertices in *Heads* or *Tails* belonging to the new S_r remain in *Heads* or *Tails* to be searched in the next iteration.

Our implementation uses the following data structure.

- A first array of size m that stores for each edge label i a set of pointers to all edges in the current graph with label i and the number of such edges; each edge of label i stores i and a reverse pointer to its location in the list of i.

- A second array of size m that stores for each label i all labels j such that $depends\_on(j) = i$.

The data structure is updated in an obvious way whenever an edge is deleted from the graph. If the number of edges with label i becomes 0, all edges with label j such that $depends\_on(j) = i$ are marked as bad. The algorithm is given on the next page. All use of the above data structure is restricted to lines 3, 4, 6(c) and 6(d), where edges are removed from G.

Lemma 3.1 *The protocol pruning algorithm is correct.*

Proof. As shown below, the while loop in the algorithm has the invariant: "There are no bad edges in G and each edge has both endpoints in some S_i. For each i, $a_i \in S_i$ and if the induced graph $G[S_i]$ is not a strongly-connected component of G then any source (sink) component of $G[S_i]$ has a vertex in

Protocol Pruning Algorithm

1. Run SCC to find the components $G[S_1], ..., G[S_l]$ of G such that $a_i \in S_i$.
2. Let m_i be the number of edges in the component $G[S_i]$.
3. Remove edges $D \cup \{uv \in E : \nexists i : \{u, v\} \subseteq S_i\}$ from G.
4. **While** there exist bad edges **do** Remove all bad edges from G.
5. Set *Heads* (*Tails*) to be the vertices in $\bigcup_i S_i$ which had an incoming (outgoing) edge just removed in line 3 or 4 above.
6. **While** $\mathit{Heads} \cup \mathit{Tails} \neq \emptyset$ **do**
   (a) Case 1: **If** $\exists i$ such that $|\mathit{Heads} \cap S_i| + |\mathit{Tails} \cap S_i| \geq \sqrt{m_i/l}$ **do**
   For each i such that $|\mathit{Heads} \cap S_i| + |\mathit{Tails} \cap S_i| \geq \sqrt{m_i/l}$:
      i. Run SCC on $G[S_i]$ to find new component $G[S_i']$ containing a_i.
      ii. Add to D all edges with at least one endpoint in $S_i \setminus S_i'$.
      iii. Remove vertices S_i from *Heads* and from *Tails* and set $S_i := S_i'$.

   (b) Case 2: **Else** For all i, $|\mathit{Heads} \cap S_i| + |\mathit{Tails} \cap S_i| < \sqrt{m_i/l}$ **do**
      i. For each $v \in \mathit{Tails}$ search G from v and for each $v \in \mathit{Heads}$ search Reverse(G) from v. Run the searches for all $1 \leq i \leq l$ in lock-step both between different $G[S_i]$ and lock-step within each $G[S_i]$ until the first search completes, say on $G[S_r]$, with all vertices R reachable from its root visited and $R \subset S_r$, but $R \neq S_r$. If no such search exists (*i.e.* $R \supseteq S_r$ for each search) HALT and output $G[S_1], ..., G[S_l]$.
      ii. If $a_r \in R$ then
         A. Add to D all edges with at least one endpoint in $S_r \setminus R$.
         B. Remove vertices S_r from *Heads* and *Tails* and set $S_r := S_r'$.
      iii. If $a_r \notin R$ then
         A. Add to D all edges with at least one endpoint in R.
         B. Remove vertices R from *Heads* and *Tails* and set $S_r := S_r \setminus R$.

   (c) Remove useless edges D from G
   (d) **While** there exist bad edges **do** Remove all bad edges from G.
   (e) Add to *Heads* (*Tails*) vertices in $\bigcup_i S_i$ which had an incoming (outgoing) edge just removed in line (c) or (d) above.
7. HALT and output $G[S_1], ..., G[S_l]$

Heads (*Tails*)." Upon termination of the while loop $Heads \cup Tails = \emptyset$ which together with the invariant ensures that there are neither any bad or useless edges left. The invariant holds initially since we delete all edges without both endpoints in some S_i in line 3 and all bad edges in line 4. Moreover, if after these edge deletions $G[S_i]$ is no longer strongly connected, then every one of its source (sink) components contains an endpoint of an edge removed, by an argument similar to one in the proof of Lemma 2.4. An execution of the loop either discovers a new component $G[S_i']$ (lines 6(a)i and 6(b)ii) or discovers some vertices R which do not belong to a strongly-connected component of a special vertex (line 6(b)iii). In the former case we remove from $Heads, Tails$ all vertices of the involved component, while in the latter case only vertices R are removed from $Heads, Tails$. In either case the relevant component vertex sets are updated and useless edges added to D. These useless edges and any subsequent bad edges are then removed from G, ensuring that the first part of the loop invariant holds. Since endpoints of removed edges are added to *Heads* and *Tails* in lines 6(c), 6(d) and 6(e) the second part of the loop invariant holds as well. Note that the algorithm may also terminate if in line 6(b)i all searches from vertices in *Heads* and *Tails* terminate in the exact same component as they were started, meaning that each $G[S_i]$ is indeed a strongly-connected component. □

Lemma 3.2 *The protocol pruning algorithm has running time $O(\sqrt{l}\sum m_i^{1.5})$.*

Proof. The cost of all data structure operations is $O(\sum_i m_i)$. For each $1 \leq i \leq l$ line 6(a)i is executed $O(\sqrt{m_i l})$ times, since each time at least $\sqrt{m_i/l}$ edges incident with the component of a_i have been deleted. An invocation of SCC costs $O(m_i)$ for a total cost of $O(\sqrt{l}\sum m_i^{1.5})$.

In an execution of line 6(b)i, if all searches terminate in the exact same component as they were started, then their total cost is $O(\sqrt{l}\sum m_i^{1.5})$, since in each $G[S_i]$ there are at most $\sqrt{m_i/l}$ searches costing $O(m_i)$ each. Otherwise, let the search initiated at $v \in S_r$ be the first one to complete, after traversing vertices R and edges E_R. Note that there are at most $\sqrt{m_r/l}$ searches in S_r. Since the parallel searches are lock-step also between different S_i, we know that for each $1 \leq i \leq l$, the parallel searches in $G[S_i]$ cost at most $O(\sqrt{m_r/l}|E_R|)$, for a total cost of $O(\sqrt{lm_r}|E_R|)$ for all searches in this execution of 6(b)i. This cost is charged to the useless edges discovered, namely those with an endpoint in $S_r \setminus R$ if $a_r \in R$ or those with an endpoint in R itself if $a_r \notin R$. It is clear that in the latter case there are at least $|E_R|$ useless edges. This also holds in the former case since R is a source (or sink) component of $G[S_r]$ and there is therefore at least one sink (or source) component of $G[S_r]$, containing useless edges, in which a parallel search had not yet completed when E_R had been traversed. Therefore there is a charge of $O(\sqrt{m_r l})$ per edge in either case. Edges are charged only once for a total cost of $O(\sqrt{l}\sum_i m_i^{1.5})$. $\square$

Acknowledgements

The authors are indebted to Moshe Vardi for suggesting the nonemptiness problem for Streett automata and to David Lee for suggesting the protocol pruning problem.

References

[1] E. A. Emerson and C. L. Lei. Modalities for model checking: Branching time strikes back. *Science of Computer Programming*, 8 (1987), 275–306.

[2] S. Even and Y. Shiloach, "An On-Line Edge-Deletion Problem", *J. ACM* 28 (1981), 1–4.

[3] R. Kurshan. Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton, 1994.

[4] D. Lee, A. N. Netravali, K. K. Sabnani. Protocol Pruning, *The Proceedings of IEEE*, October 1995.

[5] S. Safra. On the complexity of ω-automata. *Proc. 29th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1988, 319–327.

[6] R.E. Tarjan. Depth-first search and linear graph algorithms, *SIAM Journal on Computing*, vol.1, no.2, June 1972, 146-160.

[7] M. Y. Vardi and P. L. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. *Proc. 1st IEEE Symposium on Logic in Computer Science (LICS)*, 1986, 322–331.