On the Computational Complexity of Dynamic Graph Problems

G. RAMALINGAM and THOMAS REPS
University of Wisconsin–Madison

A common way to evaluate the time complexity of an algorithm is to use asymptotic worst-case analysis and to express the cost of the computation as a function of the size of the input. However, for an incremental algorithm this kind of analysis is sometimes not very informative. (By an "incremental algorithm," we mean an algorithm for a dynamic problem.) When the cost of the computation is expressed as a function of the size of the (current) input, several incremental algorithms that have been proposed run in time asymptotically no better, in the worst-case, than the time required to perform the computation from scratch. Unfortunately, this kind of information is not very helpful if one wishes to compare different incremental algorithms for a given problem.

This paper explores a different way to analyze incremental algorithms. Rather than express the cost of an incremental computation as a function of the size of the current input, we measure the cost in terms of the sum of the sizes of the *changes* in the input and the output. This change in approach allows us to develop a more informative theory of computational complexity for dynamic problems.

An incremental algorithm is said to be bounded if the time taken by the algorithm to perform an update can be bounded by some function of the sum of the sizes of the changes in the input and the output. A dynamic problem is said to be unbounded with respect to a model of computation if it has no bounded incremental algorithm within that model of computation. The paper presents new upper-bound results as well as new lower-bound results with respect to a class of algorithms called the locally persistent algorithms. Our results, together with some previously known ones, shed light on the organization of the complexity hierarchy that exists when dynamic problems are classified according to their incremental complexity with respect to locally persistent algorithms. In particular, these results separate the classes of polynomially bounded problems, inherently exponentially bounded problems, and unbounded problems.

1. INTRODUCTION

A *batch* algorithm for computing a function f is an algorithm that, given some input x, computes the output $f(x)$. (In many applications, the "input" data x is some data structure, such as a tree, graph, or matrix, while the "output" of the application, namely $f(x)$, represents some "annotation" of the x data structure—a mapping from more primitive elements that make up x, for example, graph vertices, to some space of values.) The problem of *incremental computation* is concerned with keeping the output updated as the input undergoes some changes. An *incremental algorithm* for computing f takes as input the "batch input" x, the "batch output" $f(x)$, possibly some auxiliary information, and the change in the "batch input" Δx. The algorithm computes the new "batch output" $f(x + \Delta x)$, where $x + \Delta x$ denotes the modified input, and updates the auxiliary information as necessary. A batch algorithm for computing f can obviously be used as an incremental algorithm for computing f, but often small changes in the input cause only small changes in the output and it would be more efficient to compute the new output from the old output rather than to recompute the entire output from scratch.

This work was supported in part by an IBM Graduate Fellowship, by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and CCR-9100424, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by a grant from the Digital Equipment Corporation.

Authors' addresses: G. Ramalingam, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; Thomas Reps, Computer Sciences Department, University of Wisconsin–Madison, 1210 W. Dayton St., Madison, WI 53706.
E-mail: rama@watson.ibm.com, reps@cs.wisc.edu

A common way to evaluate the computational complexity of algorithms is to use asymptotic worst-case analysis and to express the cost of the computation as a function of the size of the input. However, for incremental algorithms, this kind of analysis is sometimes not very informative. For example, when the cost of the computation is expressed as a function of the size of the (current) input, the worst-case complexity of several incremental graph algorithms is no better than that of an algorithm that performs the computation from scratch [6, 8, 19, 24, 46]. In some cases (again with costs expressed as a function of the size of the input), it has even been possible to show a lower-bound result for the problem itself, demonstrating that *no* incremental algorithm (subject to certain restrictions) for the problem can, in the worst case, run in time asymptotically better than the time required to perform the computation from scratch [3, 15, 43]. For these reasons, worst-case analysis *with costs expressed as a function of the size of the input* is often not of much help in making comparisons between different incremental algorithms.

This paper explores a different way to analyze the computational complexity of incremental algorithms. Instead of analyzing their complexity in terms of the size of the *entire* current input, we concentrate on analyzing incremental algorithms in terms of an adaptive parameter $\|\delta\|$ that captures the size of the changes in the input and output. We focus on graph problems in which the input and output values can be associated with vertices of the input graph; this lets us define CHANGED, the set of vertices whose input or output values change. We denote the number of vertices in CHANGED by $|\delta|$ and the sum of the number of vertices in CHANGED and the number of edges incident on some vertex in CHANGED by $\|\delta\|$. (A more formal definition of these parameters appears in Section 2.)

There are two very important points regarding the parameter CHANGED that we would like to be sure that the reader understands:

(1) Do not confuse CHANGED, which characterizes the amount of work that it is absolutely necessary to perform for a given dynamic *problem*, with quantities that reflect the updating costs for various internal data structures that store auxiliary information used by a particular *algorithm* for the dynamic problem. The parameter CHANGED represents the updating costs that are *inherent to the dynamic problem* itself.

(2) CHANGED is not known *a priori*. At the moment the incremental-updating process begins, only the change in the input is known. By contrast, the change in output is *unknown*—and hence so is CHANGED; both the change in the output and CHANGED are completely revealed only at the *end* of the updating process itself.

The approach used in this paper is to analyze the complexity of incremental algorithms in terms of $\|\delta\|$. An incremental algorithm is said to be *bounded* if, for all input data-sets and for all changes that can be applied to an input data-set, the time it takes to update the output solution depends only on the size of the *change* in the input and output (*i.e.*, $\|\delta\|$), and not on the size of the *entire* current input. Otherwise, an incremental algorithm is said to be *unbounded*. A problem is said to be *bounded* (unbounded) if it has (does not have) a bounded incremental algorithm. The use of $\|\delta\|$, as opposed to $|\delta|$, in the above definitions allows the complexity of a bounded algorithm to depend on the degree to which the set of vertices whose values change are connected to vertices with unchanged values. Such a dependence turns out to be natural in the problems we study.

In addition to the specific results that we have obtained on particular dynamic graph problems (see below), our work illustrates a new principle that algorithm designers should bear in mind:

Algorithms for dynamic problems can sometimes be fruitfully analyzed in terms of the parameter $\|\delta\|$

This idea represents a modest paradigm shift, and provides another arrow in the algorithm designer's quiver. The purpose of this paper is to illustrate the utility of this approach by applying it to a collection of different graph problems.

The advantage of this approach stems from the fact that the parameter $\| \delta \|$ is an *adaptive* parameter, one that varies from 1 to $|E(G)| + |V(G)|$, where $|E(G)|$ denotes the number of edges in the graph and $|V(G)|$ denotes the number of vertices in the graph. This is similar to the use of adaptive parameter $|E(G)| + |V(G)|$—which ranges from $|V(G)|$ to $|V(G)|^2$—to describe the running time of depth-first search. Note that if allowed to use only the parameter $|V(G)|$, one would have to express the complexity of depth-first search as $O(|V(G)|^2)$—which provides less information than the usual description of depth-first search as an $O(|E(G)| + |V(G)|)$ algorithm.

An important advantage of using $\| \delta \|$ is that it enables us to make distinctions between different incremental algorithms when it would not be possible to do so using worst-case analysis in terms of parameters such as $|V(G)|$ and $|E(G)|$. For instance, when the cost of the computation is expressed as a function of the size of the (current) input, all incremental algorithms that have been proposed for updating the solution to the (various versions of the) shortest-path problem after the deletion of a single edge run in time asymptotically no better, in the worst-case, than the time required to perform the computation from scratch. Spira and Pan [43], in fact, show that no incremental algorithm for the shortest path problem with positive edge lengths can do better than the best batch algorithm, under the assumption that the incremental algorithm retains only the shortest-paths information. In other words, with the usual way of analyzing incremental algorithms—worst-case analysis in terms of the size of the current input—no incremental shortest-path algorithm would appear to be any better than merely employing the best batch algorithm to recompute shortest paths from scratch! In contrast, the incremental algorithm for the problem presented in this paper is bounded and runs in time $O(\| \delta \| + |\delta| \log |\delta|)$, whereas any batch algorithm for the same problem will be an unbounded incremental algorithm.

The goal of distinguishing the time complexity of incremental algorithms from the time complexity of batch algorithms is sometimes achieved by using amortized-cost analysis. However, as Carroll observes,

> An algorithm with bad worst-case complexity will have good amortized complexity only if there is something about the problem being updated, or about the way in which we update it, or about the kinds of updates which we allow, that precludes pathological updates from happening frequently [7].

For instance, Ausiello *et al.* use amortized-cost analysis to obtain a better bound on the time complexity of a semi-dynamic algorithm they present for maintaining shortest paths in a graph as the graph undergoes a sequence of edge insertions [2]. However, in the fully dynamic version of the shortest-path problem, where both edge insertions and edge deletions are allowed, "pathological" input changes can occur frequently in a sequence of input changes. That is, when costs are expressed as a function of the size of the input, the amortized-cost complexity of algorithms for the fully dynamic version of the shortest-path problem will not, in general, be better than their worst-case complexity. Thus, the concept of boundedness permits us to distinguish between different incremental algorithms in cases where amortized analysis is of no help.

The question of amortized-cost analysis versus worst-case analysis is really orthogonal to the question studied in this paper. In the paper we demonstrate that it can be fruitful to analyze the complexity of incremental algorithms in terms of the adaptive parameter $\| \delta \|$, rather than in terms of the size of the current input. Although it happens that we use worst-case analysis in establishing all of the results presented, in principle there could exist problems for which a better bound (in terms of $\| \delta \|$) would be obtained if amortized analysis were used.

The utility of our approach is illustrated by the specific results presented in this paper:

(1) We establish several new upper-bound results: for example, the *single-sink shortest-path problem with positive edge lengths* (SSSP>0), the *all-pairs shortest-path problem with positive edge lengths* (APSP>0), and the *circuit-annotation problem* (see Section 3.2) are shown to have bounded incremental complexity. SSSP>0 and APSP>0 are shown to have $O(\| \delta \| + |\delta| \log |\delta|)$ incremental algo-

rithms; the circuit-annotation problem is shown to have an $O(2^{||\delta||})$ incremental algorithm[1].

(2) We establish several new lower-bound results, where the lower bounds are established with respect to the class of *locally persistent algorithms*, which was originally defined by Alpern *et al.* in [1]. Whereas Alpern *et al.* show the existence of a problem that has an exponential lower bound in $\| \delta \|$, we are able to demonstrate that more difficult problems exist (from the standpoint of incremental computation). In particular, we show that there are problems for which there is *no* bounded locally persistent incremental algorithm (*i.e.*, that there exist *unbounded* problems).

   We show that the class of unbounded problems contains many problems of great practical importance, such as the *closed-semiring path problems* in directed graphs and the *meet-semilattice data-flow analysis problems*.

(3) Our results, together with the results of Alpern *et al.* cited above, shed light on the organization of the complexity hierarchy that exists when dynamic problems are classified according to their incremental complexity with respect to locally persistent algorithms. In particular, these results separate the classes of polynomially bounded problems, inherently exponentially bounded problems, and unbounded problems. The computational-complexity hierarchy for dynamic problems is depicted in Figure 11. (See Section 5).

An interesting aspect of this complexity hierarchy is that it separates problems that, at first glance, are apparently very similar. For example, SSSP>0 is polynomially bounded, yet the very similar problem SSSP≥0 (in which *0-length* edges are also permitted) is unbounded. Some other related results have been left out of this paper due to length considerations, including a generalization of the above-mentioned lower bound proofs to a much more powerful model of computation than the class of locally persistent algorithms, and a generalization of the incremental algorithm for the shortest-path problem to a more general class of problems. (See [29].)

The remainder of the paper is organized into five sections. Section 2 introduces terminology and notation. Section 3 presents bounded incremental algorithms for three problems: SSSP>0, APSP>0, and the circuit-annotation problem. Section 4 concerns lower-bound results, where lower bounds are established with respect to locally persistent algorithms. The results from Sections 3 and 4, together with some previously known results, shed light on the organization of the complexity hierarchy that exists when incremental-computation problems are classified according to their incremental complexity with respect to locally persistent algorithms. This complexity hierarchy is presented in Section 5. Section 6 discusses how the results reported in this paper relate to previous work on incremental computation and incremental algorithms.

2. Terminology

We now formulate a notion of the "size of the change in input and output" that is applicable to the class of graph problems in which the input consists of a graph G, and possibly some information (such as a real value) associated with each vertex or edge of the graph, and the output consists of a value $S_G(u)$ for each vertex u of the graph G. (For instance, in SSSP>0, $S_G(u)$ is the length of the shortest path from vertex u to a distinguished vertex, denoted by $sink(G)$.) Thus, each vertex/edge in the graph may have an associated *input* value, and each vertex in the graph has an associated *output* value.

[1]This complexity measure holds for the circuit-annotation problem under certain assumptions explained in Section 3.3. Under less restricted assumptions, the circuit-annotation problem has an $O(\| \delta \| 2^{||\delta||})$ incremental algorithm [29].

A **directed graph** $G = (V(G), E(G))$ consists of a set of **vertices** $V(G)$ and a set of **edges** $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. An edge $(b,c) \in E(G)$, where b, $c \in V(G)$, is said to be directed from b to c, and will be more mnemonically denoted by $b \longrightarrow c$. We say that b is the **source** of the edge, that c is the **target**, that b is a **predecessor** of c, and that c is a **successor** of b. A vertex b is said to be **adjacent** to a vertex c if b is a successor or predecessor of c. The set of all successors of a vertex a in G is denoted by $\boldsymbol{Succ_G(a)}$, while the set of all predecessors of a in G is denoted by $\boldsymbol{Pred_G(a)}$. If K is a set of vertices, then $\boldsymbol{Succ_G(K)}$ denotes $\bigcup_{a \in K} Succ_G(a)$, and $\boldsymbol{Pred_G(K)}$ is similarly defined. Given a set K of vertices in a graph G, the **neighborhood** of K, denoted by $\boldsymbol{N_G(K)}$, is defined be the set of all vertices that are in K or are adjacent to some vertex in K: $N_G(K) = K \cup Succ_G(K) \cup Pred_G(K)$. The set $N^i_G(K)$ is defined inductively to be $N_G(N^{i-1}_G(K))$, where $N^0_G(K) = K$.

For any set of vertices K, we will denote the cardinality of K by both $|K|$ and V_K. For our purposes, a more useful measure of the "size" of K is the *extended size* of K, which is defined as follows: Let E_K be the number of edges that have at least one endpoint in K. The **extended size** of K (of order 1), denoted by $\|K\|_{1,G}$ or just $\|K\|$, is defined to be $V_K + E_K$. In other words, $\|K\|$ is the sum of the number of vertices in K and the number of edges with an endpoint in K. The extended size of K of order i, denoted by $\|K\|_{i,G}$ or just $\|K\|_i$, is defined to be $V_{N^{i-1}(K)} + E_{N^{i-1}(K)}$—in other words, it is the extended size of $N^{i-1}(K)$. In this paper, we are only ever concerned with the extended size of order 1, except in a couple of places where the extended size of order 2 is required.

We restrict our attention to "unit changes": changes that modify the information associated with a single vertex or edge, or that add or delete a single vertex or edge. We denote by $G+\delta$ the graph obtained by making a change δ to graph G. A vertex u in G or $G+\delta$ is said to have been **modified** by δ if δ inserted or deleted u, or modified the input value associated with u, or inserted or deleted some edge incident on u, or modified the information associated with some edge incident on u. The set of all modified vertices in $G+\delta$ will be denoted by $\mathrm{MODIFIED}_{G,\delta}$. Note that this set captures the change in the input. A vertex in $G+\delta$ is said to be an **affected** vertex either if it is a newly inserted vertex or if its output value in $G+\delta$ is different from its output value in G. Let $\mathrm{AFFECTED}_{G,\delta}$ denote the set of all affected vertices in $G+\delta$. This set captures the change in the output. We define $\mathrm{CHANGED}_{G,\delta}$ to be $\mathrm{MODIFIED}_{G,\delta} \cup \mathrm{AFFECTED}_{G,\delta}$. This set, which we occasionally abbreviate further to just δ, captures the change in the input *and* output. The subscripts of the various terms defined above will be dropped if no confusion is likely.

We use $\|\mathrm{MODIFIED}\|_{i,G+\delta}$ as a measure of the size of the change in input, $\|\mathrm{AFFECTED}\|_{i,G+\delta}$ as a measure of the size of the change in output, and $\|\mathrm{CHANGED}\|_{i,G+\delta}$, which we abbreviate to $\|\delta\|_i$, as a measure of the size of the change in the input and output. An omitted subscript i implies a value of 1.

In summary, $|\delta|$ denotes V_δ, the number of vertices that are modified or affected, while $\|\delta\|$ denotes $V_\delta + E_\delta$, where E_δ is the number of edges that have at least one endpoint that is modified or affected.

An incremental algorithm for a problem P takes as input a graph G, the solution to graph G, possibly some auxiliary information, and input change δ. The algorithm computes the solution for the new graph $G+\delta$ and updates the auxiliary information as necessary. The time taken to perform this update step may depend on G, δ, and the auxiliary information. An incremental algorithm is said to be *bounded* if, for a fixed value of i, we can express the time taken for the update step entirely as a function of the parameter

$\| \delta \|_{i,G}$ (as opposed to other parameters, such as $|V(G)|$ or $|G|$).[2] It is said to be *unbounded* if its running time can be arbitrarily large for fixed $\| \delta \|_{i,G}$. A problem is said to be bounded (unbounded) if it has (does not have) a bounded incremental algorithm.

3. Upper-bound Results: Three Bounded Dynamic Problems

This section concerns three new upper-bound results. In particular, bounded incremental algorithms are presented for the single-sink shortest-path problem with positive edge weights (SSSP>0), the all-pairs shortest-path problem with positive edge weights (APSP>0), and the circuit-annotation problem. SSSP>0 and APSP>0 are shown to be polynomially bounded; the circuit-annotation problem is shown to be exponentially bounded.

3.1. The Incremental Single-Sink Shortest-Path Problem

The input for SSSP>0 consists of a directed graph G with a distinguished vertex $sink(G)$. Every edge $u \longrightarrow v$ in the graph has a positive real-valued length, which we denote by $length(u \longrightarrow v)$. The length of a path is defined to be the sum of the lengths of the edges in the path. We are interested in computing $dist(u)$, the length of the shortest path from u to $sink(G)$, for every vertex u in the graph. If there is no path from a vertex u to $sink(G)$ then $dist(u)$ is defined to be infinity.

This section concerns the problem of updating the solution to an instance of the SSSP>0 problem after a unit change is made to the graph. The insertion or deletion of an isolated vertex can be processed trivially and will not be discussed here. We present algorithms for performing the update after a single edge is deleted from or inserted into the edge set of G. The operations of inserting an edge and decreasing the length of an edge are equivalent in the following sense: The insertion of an edge can be considered as the special case of an edge length being decreased from ∞ to a finite value, while the case of a decrease in an edge length can be considered as the insertion of a new edge parallel to the relevant edge. The operations of deleting an edge and increasing an edge length are similarly equivalent. Consequently, the algorithms we present here can be directly adapted for performing the update after a change in the length of an edge.

Proposition 1. *SSSP>0 has a bounded incremental algorithm. In particular, there exists an algorithm* $\text{DeleteEdge}_{\text{SSSP>0}}$ *that can process the deletion of an edge in time* $O(\| \delta \| + |\delta| \log |\delta|)$ *and there exists an algorithm* $\text{InsertEdge}_{\text{SSSP>0}}$ *that can process the insertion of an edge in time* $O(\| \delta \| + |\delta| \log |\delta|)$.

Though we have defined the incremental SSSP>0 problem to be that of maintaining the *lengths* of the shortest paths to the sink, the algorithms we present maintain the shortest paths as well. An edge in the graph is said to be an *SP* edge iff it occurs on some shortest path to the sink. Thus, an edge $u \longrightarrow v$ is an *SP* edge iff $dist(u) = length(u \longrightarrow v) + dist(v)$. A subgraph T of G is said to be a (single-sink) shortest-paths tree for the given graph G with sink $sink(G)$ if (i) T is a (directed) tree rooted at $sink(G)$, (ii) $V(T)$ is the set of all vertices that can reach $sink(G)$ in G, and (iii) every edge in T is an *SP* edge. Thus, for every vertex u in $V(T)$, the unique path in T from u to $sink(G)$ is a shortest path.

The set of all *SP* edges of the graph, which we denote by $SP(G)$, induces a subgraph of the given graph, which we call the shortest-paths subgraph. We will occasionally denote the shortest-paths *subgraph* also by $SP(G)$. Note that a path from some vertex u to the sink vertex is a shortest path iff it occurs in $SP(G)$ (*i.e.*, iff all the edges in that path occur in $SP(G)$). Since all edges in the graph are assumed to have a positive length, any shortest path in the graph must be acyclic. Consequently, $SP(G)$ is a directed acyclic

[2]Note that we use the uniform-cost measure in analyzing the complexity of the steps of an algorithm.

graph (DAG). As we will see later, this is what enables us to process input changes in a bounded fashion. If zero length edges are allowed, then $SP(G)$ can have cycles, and the algorithms we present in this section will not work correctly in all instances.

Our incremental algorithm for SSSP>0 works by maintaining the shortest-path subgraph $SP(G)$. We will also find it useful to maintain the outdegree of each vertex u in the subgraph $SP(G)$.

3.1.1. Deletion of an Edge

The update algorithm for edge deletion is given as procedure DeleteEdge$_{\text{SSSP>0}}$ in Figure 1.

We will find it useful in the following discussion to introduce the concept of an *affected edge*. An *SP* edge $x \longrightarrow y$ is said to be affected by the deletion of the edge $v \longrightarrow w$ if there exists no path in the new graph from x to the sink that makes use of the edge $x \longrightarrow y$ *and* has a length equal to $dist_{old}(x)$. It is easily seen that $x \longrightarrow y$ is an affected *SP* edge iff y is an affected vertex. On the other hand, any vertex x other than v (the source of the deleted edge) is an affected vertex iff all *SP* edges going out of x are affected edges. The vertex v itself is an affected vertex iff $v \longrightarrow w$ is the only *SP* edge going out of vertex v.

The algorithm for updating the solution (and $SP(G)$) after the deletion of an edge works in two phases. The first phase (lines [4]–[14]) computes the set of all affected vertices and affected edges and removes the affected edges from $SP(G)$, while the second phase (lines [15]–[30]) computes the new output value for all the affected vertices and updates $SP(G)$ appropriately.

Phase 1: Identifying affected vertices

A vertex's *dist* value increases due to the deletion of edge $v \longrightarrow w$ iff all shortest paths from the vertex to $sink(G)$ make use of edge $v \longrightarrow w$. In other words, if $SP(G)$ denotes the *SP* DAG of the original graph, then the set of affected vertices is precisely the set of vertices that can reach the sink in $SP(G)$ but not in $SP(G) - \{v \longrightarrow w\}$, the DAG obtained by deleting edge $v \longrightarrow w$ from $SP(G)$.

Thus, Phase 1 is essentially an incremental algorithm for the single-sink reachability problem in DAGs that updates the solution after the deletion of an edge. The algorithm is very similar to the topological sorting algorithm. It maintains a set of vertices (WorkSet) that have been identified as being affected but have not yet been processed. Initially v is added to this set if $v \longrightarrow w$ is the only *SP* edge going out of v. The vertices in WorkSet are processed one by one. When a vertex u is processed, all *SP* edges coming into u are removed from $SP(G)$ since they are affected edges. During this process some vertices may be identified as being affected (because there no longer exists any *SP* edge going out of those vertices) and may be added to the workset.

We maintain $outdegree_{SP}(x)$, the number of *SP* edges going out of vertex x, so that the tests in lines [3] and [12] can be performed in constant time. We have not discussed how the subgraph $SP(G)$ is maintained. If $SP(G)$ is represented by maintaining (adjacency) lists at each vertex of all incoming and outgoing *SP* edges, then it is not necessary to maintain $outdegree_{SP}(x)$ separately, since $outdegree_{SP}(x)$ is zero iff the list of outgoing SP edges is empty. Alternatively, we can save storage by not maintaining $SP(G)$ explicitly. Given any edge $x \longrightarrow y$, we can check if that edge is in $SP(G)$ in constant time, by checking if $dist(x) = length(x \longrightarrow y) + dist(y)$. In this case, however, it is necessary to maintain $outdegree_{SP}(x)$ or else the cost of Phase 1 increases to $O(\| \delta \|_2)$.

We now analyze the time complexity of Phase 1. The loop in lines [7]–[14] performs exactly $|\text{AFFECTED}|$ iterations, once for each affected vertex u. The iteration corresponding to vertex u takes time $O(|Pred(u)|)$. Consequently, the running time of Phase 1 is $O(\sum_{u \in \text{AFFECTED}} |Pred(u)|) = O(\| \text{AFFECTED} \|)$. If we choose to maintain the *SP* DAG explicitly, then the running time is actually linear in the extended size of AFFECTED in the *SP* DAG, which can be less than the extended size of AFFECTED in the graph G itself.

```
procedure DeleteEdge_{SSSP>0}($G$, $v \longrightarrow w$)
declare
        $G$: a directed graph;
        $v \longrightarrow w$: an edge to be deleted from $G$
        WorkSet, AffectedVertices: sets of vertices;
        PriorityQueue: a heap of vertices
        $a, b, c, u, v, w, x, y$: vertices
preconditions
        $SP(G)$ is the shortest-paths subgraph of $G$
        $\forall v \in V(G)$, $outdegree_{SP}(v)$ is the outdegree of vertex $v$ in the shortest-paths subgraph $SP(G)$
        $\forall v \in V(G)$, $dist(v)$ is the length of the shortest path from $v$ to $sink(G)$
begin
[1]     if $v \longrightarrow w \in SP(G)$ then
[2]        Remove edge $v \longrightarrow w$ from $SP(G)$ and from $E(G)$ and decrement $outdegree_{SP}(v)$
[3]        if $outdegree_{SP}(v) = 0$ then
[4]          /* Phase 1: Identify the affected vertices and remove the affected edges from $SP(G)$ */
[5]           WorkSet := { $v$ }
[6]           AffectedVertices := $\emptyset$
[7]           while WorkSet $\neq \emptyset$ do
[8]              Select and remove a vertex $u$ from WorkSet
[9]              Insert vertex $u$ into AffectedVertices
[10]             for every vertex $x$ such that $x \longrightarrow u \in SP(G)$ do
[11]                Remove edge $x \longrightarrow u$ from $SP(G)$ and decrement $outdegree_{SP}(x)$
[12]                if $outdegree_{SP}(x) = 0$ then Insert vertex $x$ into WorkSet fi
[13]             od
[14]          od
[15]         /* Phase 2: Determine new distances from affected vertices to $sink(G)$ and update $SP(G)$. */
[16]          PriorityQueue := $\emptyset$
[17]          for every vertex $a \in$ AffectedVertices do
[18]             $dist(a)$ := min ({ $length(a \longrightarrow b) + dist(b)$ |
                                        $a \longrightarrow b \in E(G)$ and $b \notin$ AffectedVertices) } $\cup$ { $\infty$ })
[19]             if $dist(a) \neq \infty$ then InsertHeap(PriorityQueue, $a$, $dist(a)$) fi
[20]          od
[21]          while PriorityQueue $\neq \emptyset$ do
[22]             $a$ := FindAndDeleteMin(PriorityQueue)
[23]             for every vertex $b \in Succ(a)$ such that $length(a \longrightarrow b) + dist(b) = dist(a)$ do
[24]                Insert edge $a \longrightarrow b$ into $SP(G)$ and increment $outdegree_{SP}(a)$
[25]             od
[26]             for every vertex $c \in Pred(a)$ such that $length(c \longrightarrow a) + dist(a) < dist(c)$ do
[27]                $dist(c)$ := $length(c \longrightarrow a) + dist(a)$
[28]                AdjustHeap( PriorityQueue, $c$, $dist(c)$)
[29]             od
[30]          od
[31]       fi
[32]    else Remove edge $v \longrightarrow w$ from $E(G)$
[33]    fi
end
postconditions
        $SP(G)$ is the shortest-paths subgraph of $G$
        $\forall v \in V(G)$, $outdegree_{SP}(v)$ is the outdegree of vertex $v$ in the shortest-paths subgraph $SP(G)$
        $\forall v \in V(G)$, $dist(v)$ is the length of the shortest path from $v$ to $sink(G)$
```

Figure 1. An algorithm to update the SSSP>0 solution and $SP(G)$ after the deletion of an edge.

Phase 2: Determining new distances for affected vertices and updating $SP(G)$

Phase 2 of DeleteEdge$_{\text{SSSP>0}}$ is an adaptation of Dijkstra's batch shortest-path algorithm that uses priority-first search [42] to compute the new *dist* values for the affected vertices.

Consider Figure 2. Assume that for every vertex y in set A the length of the shortest path from y to the sink is known and is given by $dist(y)$. We need to compute the length of the shortest path from x to the sink for every vertex x in the set of remaining vertices, B. Consider the graph obtained by "condensing" A to a new sink vertex: that is, we replace the set of vertices A by a new sink vertex s, and replace every edge $x \longrightarrow y$ from a vertex x in B to a vertex y in A by an edge $x \longrightarrow s$ of length $length(x \longrightarrow y) + dist(y)$. The given problem reduces to the SSSP problem for this reduced graph, which can be solved using Dijkstra's algorithm. Phase 2 of our algorithm works essentially this way.

Before we analyze the complexity of Phase 2, we explain the heap operations we make use of in the algorithm. The operation $InsertHeap(H,i,k)$ inserts an item i into heap H with a key k. The operation $FindAndDeleteMin(H)$ returns the item in heap H that has the minimum key and deletes it from the heap. The operation $AdjustHeap(H,i,k)$ inserts an item i into *Heap* with key k if i is not in *Heap*, and changes the key of item i in *Heap* to k if i is in *Heap*. In this algorithm, *AdjustHeap* either inserts an item into the heap,

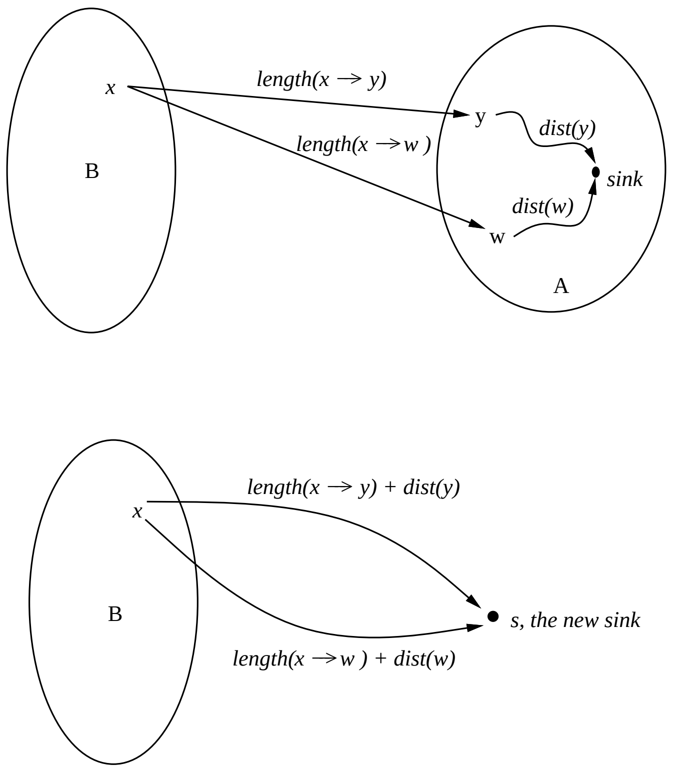


Figure 2. Phase 2 of DeleteEdge$_{\text{SSSP>0}}$. Let A be the set of unaffected vertices and let B be the set of affected vertices. The correct *dist* value is known for every vertex in *A* and the new *dist* value has to be computed for every vertex in *B*. This problem can be reduced to a *batch instance* of the SSSP>0 problem, namely the SSSP>0 problem for the graph obtained as follows: we take the subgraph induced by the set B of vertices, introduce a new sink vertex, and for every edge $x \longrightarrow y$ from a vertex in B to a vertex outside *B*, we add an edge from x to the new sink vertex, with length $length(x \longrightarrow y) + dist(y)$.

or decreases the key of an item in the heap.

The complexity of Phase 2 depends on the type of heap we use. We assume that PriorityQueue is implemented as a relaxed heap (see [12]). Both insertion of an item into a relaxed heap and decreasing the key of an item in a relaxed heap cost $O(1)$ time, while finding and deleting the item with the minimum key costs $O(\log p)$ time, where p is the number of items in the heap.

The loop in lines [21]-[30] iterates at most $|\text{AFFECTED}|$ times. An affected vertex a is processed in each iteration, but not all affected vertices may be processed. In particular, affected vertices that can no longer reach the sink vertex will not be processed. Each iteration takes $O(\| \{a\} \|)$ time for lines [23]-[29], and $O(\log |\text{AFFECTED}|)$ time for the heap operation in line [22]. Hence, the running time of Phase 2 is $O(\| \text{AFFECTED} \| + |\text{AFFECTED}| \log |\text{AFFECTED}|)$.

It follows from the bounds on the running time of Phase 1 and Phase 2 that the total running time of DeleteEdge$_{\text{SSSP>0}}$ is bounded by $O(\| \text{AFFECTED} \| + |\text{AFFECTED}| \log |\text{AFFECTED}|)$, which is $O(\| \delta \| + |\delta| \log |\delta|)$.

3.1.2. Insertion of an Edge

We now turn to the problem of updating distances and the set $SP(G)$ after an edge $v \longrightarrow w$ with length c is inserted into G. The algorithm for this problem, procedure InsertEdge$_{\text{SSSP>0}}$, is presented in Figure 4. (The algorithm presented works correctly even if the length of the newly inserted edge is non-positive as long as all edges in the original graph have a positive length and the new edge does not introduce a cycle of negative length. This will be important in generalizing our incremental algorithm to handle edges of non-positive lengths. See Section 3.1.3.)

The algorithm is based on the following characterization of the region of affected vertices, which enables the updating to be performed in a bounded fashion. If the insertion of edge $v \longrightarrow w$ causes u to be an affected vertex, then any new shortest path from u to $sink(G)$ must consist of a shortest path from u to v, followed by the edge $v \longrightarrow w$, followed by a shortest path from w to $sink(G)$. In particular, a vertex u is affected iff $dist(u,v) + length(v \longrightarrow w) + dist_{old}(w) < dist_{old}(u)$, where $dist(u,v)$ is the length of the shortest path from u to v in the new graph, and $dist_{old}$ refers to the lengths of the shortest paths to the sink in the graph before the insertion of the edge $v \longrightarrow w$. The new $dist$ value for an affected vertex u is given by $dist(u,v) + length(v \longrightarrow w) + dist_{old}(w)$.

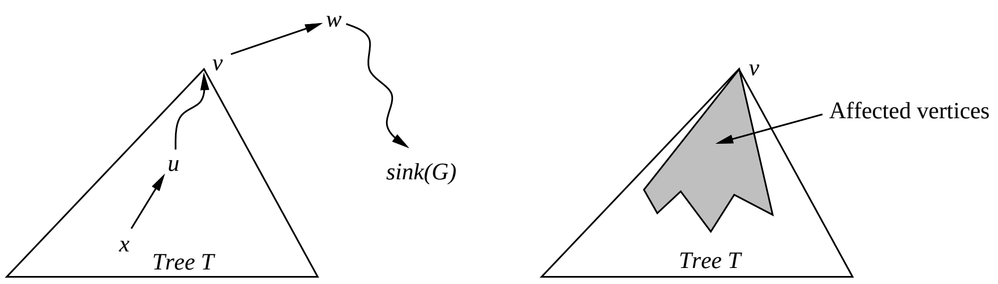


Figure 3. T is a shortest-path tree for sink v. If x is an affected vertex, then u, the parent of x in T, must also be an affected vertex. Hence, the set of all vertices affected by the insertion of the edge $v \longrightarrow w$ forms a connected subtree at the root of T.

Consider T, a single-sink shortest-path tree for the vertex v. Let x be any vertex, and let u be the parent of x in T. (See Figure 3.) If x is an affected vertex, then u must also be an affected vertex: otherwise, there must exist some shortest path P from u to $sink(G)$ that does not contain edge $v \longrightarrow w$; the path consisting of the edge $x \longrightarrow u$ followed by P is then a shortest path from x to $sink(G)$ that does not contain edge $v \longrightarrow w$; hence, x cannot be an affected vertex, contradicting our assumption. In other words, any ancestor (in T) of an affected vertex must also be an affected vertex. The set of all affected vertices must, hence, form a connected subtree of T at the root of T.

The algorithm works by using an adaptation of Dijkstra's algorithm to construct the part of the tree T restricted to the affected vertices (the shaded part of T in Figure 3) in lines [3]-[6], [10], [11], and [16]-[19]. As in Dijkstra's algorithm, the keys of vertices in PriorityQueue indicate distances from u to v. However, unlike in Dijkstra's algorithm, these distances are available only indirectly; the distance annotation at u (*i.e.*, $dist(u)$) indicates the distance from u to $sink(G)$, not that from u to v. Appropriate adjustments are made in line [6]—the key for vertex v is 0—and in line [19]—the key for vertex u is $dist(x) - dist(u)$.

When the vertex u is selected from PriorityQueue in line [11], its priority is nothing but $dist(u,v)$. In a normal implementation of Dijkstra's algorithm, every predecessor x of u would then be examined (as in the loop in lines [16]-[23]), and its priority in PriorityQueue would be adjusted if $length(x \longrightarrow u)$ + $dist(u,v)$ was less than the length of the shortest path found so far from x to v. Here, we instead adjust the priority of x or insert it into PriorityQueue *only if* $length(x \longrightarrow u)$ + $dist(u)$ is less than $dist(x)$: that is, only if edge $x \longrightarrow u$ followed by a shortest path from u to $sink(G)$ yields a path shorter than the shortest path currently known from x to $sink(G)$. In other words, a vertex x is added to PriorityQueue only if it is an affected vertex. In effect, the algorithm avoids constructing the unshaded part of the tree T in Figure 3.

During this process, the set of all affected vertices is identified and every affected vertex is assigned its correct value finally. If v is affected, it is assigned its correct value in line [5]; any other affected vertex x will be assigned its correct value in line [18]. Simultaneously, the algorithm also updates the set of edges $SP(G)$ as follows. If v is unaffected but $v \longrightarrow w$ becomes an SP edge, it is added to $SP(G)$ in line [8]. Similarly any edge $x \longrightarrow u$ that becomes an SP edge, while x is unaffected, is identified and added to $SP(G)$ in line [21]. For any affected vertex u, an edge $u \longrightarrow x$ directed away from u can change its SP edge status. These changes are identified and made to $SP(G)$ in lines [12]-[15].

Note that unlike procedure DeleteEdge$_{\text{SSSP}>0}$, in which the process of identifying which vertices are members of AFFECTED and the process of updating *dist* values are separated into separate phases, in procedure InsertEdge$_{\text{SSSP}>0}$ the identification of AFFECTED is *interleaved* with updating. Observe, too, that the algorithm works correctly even if the length of the newly inserted edge is negative, as long as all other edges have a positive length and the new edge does not introduce a cycle of negative length. The reason is that we require edges to have a non-negative length only in the (partial) construction of the tree T. But in constructing a shortest-path tree for some sink vertex, one can always ignore edges going *out of the sink vertex*, as long as there are no negative length cycles. Consequently, it is immaterial, in the construction of T, whether $length(v \longrightarrow w)$ is negative or not.

We now analyze the time complexity of InsertEdge$_{\text{SSSP}>0}$. The loop in lines [10]-[24] iterates once for every affected vertex u. Each iteration takes time $O(\log |\text{AFFECTED}|)$ for line [11] and time $O(\| \{u\} \|)$ for lines [12]-[23]. Note that the AdjustHeap operation in line [19] either inserts a vertex into the heap or decreases the key of a vertex in the heap. Hence it costs only $O(1)$ time. Thus, the running time of procedure InsertEdge$_{\text{SSSP}>0}$ is $O(\| \text{AFFECTED} \| + |\text{AFFECTED}| \log |\text{AFFECTED}|)$, which is $O(\| \delta \| + |\delta| \log |\delta|)$.

```
procedure InsertEdge_{SSSP>0}$(G, v \longrightarrow w, c)$
declare
        $G$: a directed graph
        $v \longrightarrow w$: an edge to be inserted in $G$
        $c$: a positive real number indicating the length of edge $v \longrightarrow w$
        PriorityQueue: a heap of vertices
preconditions
        $SP(G)$ is the shortest-paths subgraph of $G$
        $\forall v \in V(G)$, $outdegree_{SP}(v)$ is the outdegree of vertex $v$ in the shortest-paths subgraph $SP(G)$
        $\forall v \in V(G)$, $dist(v)$ is the length of the shortest path from $v$ to $sink(G)$
begin
[1]     Insert edge $v \longrightarrow w$ into $E(G)$
[2]     $length(v \longrightarrow w) := c$
[3]     PriorityQueue := $\varnothing$
[4]     if $length(v \longrightarrow w) + dist(w) < dist(v)$ then
[5]        $dist(v) := length(v \longrightarrow w) + dist(w)$
[6]        InsertHeap(PriorityQueue, $v$, 0)
[7]     else if $length(v \longrightarrow w) + dist(w) = dist(v)$ then
[8]        Insert $v \longrightarrow w$ into $SP(G)$ and increment $outdegree_{SP}(v)$
[9]     fi
[10]    while PriorityQueue $\neq \varnothing$ do
[11]       $u$ := FindAndDeleteMin(PriorityQueue)
[12]       Remove all edges of $SP(G)$ directed away from $u$ and set $outdegree_{SP}(u) = 0$
[13]       for every vertex $x \in Succ(u)$ do
[14]          if $length(u \longrightarrow x) + dist(x) = dist(u)$ then Insert $u \longrightarrow x$ into $SP(G)$ and increment $outdegree_{SP}(u)$ fi
[15]       od
[16]       for every vertex $x \in Pred(u)$ do
[17]          if $length(x \longrightarrow u) + dist(u) < dist(x)$ then
[18]             $dist(x) := length(x \longrightarrow u) + dist(u)$
[19]             AdjustHeap(PriorityQueue, $x$, $dist(x) - dist(v)$)
[20]          else if $length(x \longrightarrow u) + dist(u) = dist(x)$ then
[21]             Insert $x \longrightarrow u$ into $SP(G)$ and increment $outdegree_{SP}(x)$
[22]          fi
[23]       od
[24]    od
end
postconditions
        $SP(G)$ is the shortest-paths subgraph of $G$
        $\forall v \in V(G)$, $outdegree_{SP}(v)$ is the outdegree of vertex $v$ in the shortest-paths subgraph $SP(G)$
        $\forall v \in V(G)$, $dist(v)$ is the length of the shortest path from $v$ to $sink(G)$
```

Figure 4. An algorithm to update the SSSP>0 solution and $SP(G)$ after the insertion of an edge $v \longrightarrow w$ into graph G.

3.1.3. Incremental Updating in the Presence of Negative Edge-Lengths

We now briefly discuss the problem of updating the solution to the single-sink shortest-path problem in the presence of edges of non-positive lengths. The obstacle to obtaining a bounded incremental algorithm for this generalized problem is the presence of cycles of length zero, and not edges of negative lengths. We show in Section 4.2 that there exists *no* bounded locally persistent incremental algorithm for maintaining shortest paths if 0-length cycles are allowed in the graph. However, bounded locally persistent incremental algorithms *do* exist for the dynamic SSSP-Cycle>0 problem: the single-sink shortest-path problem in graphs where edges may have arbitrary length but all cycles have positive length.

The algorithms DeleteEdge$_{\text{SSSP>0}}$ and InsertEdge$_{\text{SSSP>0}}$ work correctly even in the presence of 0-length *edges* as long as there are no 0-length *cycles*. These algorithms do not work correctly in the presence of negative-length edges for the same reasons that Dijkstra’s algorithm does not. However, a simple modification to DeleteEdge$_{\text{SSSP>0}}$ yields an algorithm for updating the solution to the SSSP-Cycle>0 problem after the deletion of an edge (with no change in the time-complexity). A similar modification to InsertEdge$_{\text{SSSP>0}}$ yields an algorithm for updating the solution to the SSSP-Cycle>0 problem after the insertion of an edge $u \longrightarrow v$, as long as the sink vertex was already reachable from vertex u. These generalizations are based on the technique of Edmonds and Karp for transforming the length of every edge in a graph to a non-negative real without changing the graph's shortest paths [13, 44], and are described in [28, 29].

The above techniques for updating the solution to the SSSP-Cycle>0 problem fail for only one type of input change, namely the insertion of an edge $u \longrightarrow v$ that creates a path from u to the sink vertex where no path existed before. However, even such an input modification can be handled in time $O(\|\delta\| \cdot |\delta|)$ by using an adaptation of the Bellman-Ford algorithm for the shortest-paths problem. (See [29].)

3.2. The Dynamic All-Pairs Shortest-Path Problem

This section concerns a bounded incremental algorithm for a version of the dynamic all-pairs shortest-path problem with positive-length edges (APSP>0).

We will assume that the vertices of G are indexed from $1..|V(G)|$. APSP>0 involves computing the entries of a *distance matrix, dist*$[1..|V(G)|, 1..|V(G)|]$, where entry *dist*$[i, j]$ represents the length of the shortest path in G from vertex i to vertex j. It is also useful to think of this information as being associated with the individual vertices of the graph: with each vertex there is an *array* of values, indexed from $1..|V(G)|$—the j^{th} value at vertex i records the length of the shortest path in G from vertex i to vertex j. This lets us view the APSP>0 problem as a graph problem that requires the computation of an output value for each vertex in the graph. However, APSP>0 does not fall into the class of graph problems that involve the computation of a *single atomic* value for each vertex u in the input graph, and so, as explained below, some of our terminology in this section differs from the terminology that was introduced in Section 2.

Since MODIFIED measures the change in the input, the definition of MODIFIED remains the same (and hence for a single-edge change to the graph $|\text{MODIFIED}| = 2$). In order to define AFFECTED, which measures the change in the output, we view the problem as n instances of the SSSP>0 problem. Let AFFECTED$_u$ represent the set of affected vertices for the single-sink problem with u as the sink vertex. We define $|\text{AFFECTED}|$ for the APSP>0 problem as follows:

$$|\text{AFFECTED}| = \sum_{u=1}^{|V(G)|} |\text{AFFECTED}_u|.$$

Thus, $|\text{AFFECTED}|$ is the number of entries in the *dist* matrix that change in value. We define the extended size $\|\text{AFFECTED}\|$ as follows:

$$\|\text{AFFECTED}\|_i = \sum_{u=1}^{|V(G)|} \|\text{AFFECTED}_u\|_i,$$

Note that for a given change δ, some or all of the AFFECTED$_u$ can be empty and, hence, $\|\text{AFFECTED}\|_i$ may be less than $|V(G)|$. The parameter $\|\delta\|_i$ in which we measure the incremental complexity of APSP>0 is defined as follows:

$$\|\delta\|_i = \|\text{MODIFIED}\|_i + \|\text{AFFECTED}\|_i.$$

The parameter $|\delta|$ is also similarly defined.

The definitions of AFFECTED, $\|\text{AFFECTED}\|_i$, and $\|\delta\|_i$ given above are clearly in the same spirit as those from Section 2.

We now turn our attention to the problem of updating the solution to an instance of the APSP>0 problem after a unit change.

The operations of inserting and deleting isolated vertices are trivially handled but for some concerns having to do with dynamic storage allocation. Whether the shortest-path distances are stored in a single two-dimensional array or in a collection of one-dimensional arrays, we face the need to *increase* or *decrease* the array size(s). We can do this by dynamically expanding and contracting these arrays using the well-known doubling/halving technique (see Section 18.4 of [10], for example). Assume the distance matrix is maintained as a collection of *n* vectors (of equal size), where *n* is the number of vertices in the graph. Whenever a new vertex is inserted, a new vector is allocated. Whenever the number of vertices in the graph exceeds the size of the individual vectors, the size of each of the vectors is doubled (by re-allocation). Vertex deletion is similarly handled, by halving the size of the vectors when appropriate. The insertion or deletion of an isolated vertex has an *amortized* cost of $O(|V(G)|)$ under this scheme: doubling or halving the arrays takes time $O(|V(G)|^2)$, but the cost is amortized over $\Omega(|V(G)|)$ vertex insertion/deletion operations. A cost of $O(|V(G)|)$ is reasonable, in the sense that the introduction or removal of an isolated vertex causes $O(|V(G)|)$ "changes" to entries in the distance matrix. Thus, in some sense for such operations $|\delta| = \Theta(|V(G)|)$, and hence the amortized cost of the doubling/halving scheme is optimal.

We now consider the problem of updating the solution after the insertion or deletion of an edge. As explained in the previous section, it is trivial to generalize these operations to handle the shortening or lengthening of an edge, respectively.

Proposition 2. *APSP>0 has a bounded incremental algorithm. In particular, there exists an algorithm* $\text{DeleteEdge}_{\text{APSP>0}}$ *that can process an edge deletion in time* $O(\|\delta\|_2 + |\delta| \log |\delta|)$, *and there exists an algorithm* $\text{InsertEdge}_{\text{APSP>0}}$ *that can process an edge insertion in time* $O(\|\delta\|_1)$.

3.2.1. Deletion of an Edge

The basic idea behind the bounded incremental algorithm for $\text{DeleteEdge}_{\text{APSP>0}}$ is to make repeated use of the bounded incremental algorithm $\text{DeleteEdge}_{\text{SSSP>0}}$ as a subroutine, but with a different sink vertex on each call. A simple incremental algorithm for $\text{DeleteEdge}_{\text{APSP>0}}$ would be to make as many calls on $\text{DeleteEdge}_{\text{SSSP>0}}$ as there are vertices in graph *G*. However, this method is not bounded because it would perform at least *some* work for each vertex of *G*; the total updating cost would be at least $\Omega(|V(G)|)$, which in general is not a function of $\|\delta\|_i$ for any fixed value of *i*.

The key observation behind our bounded incremental algorithm for $\text{DeleteEdge}_{\text{APSP>0}}$ is that it is possible to determine *exactly* which calls on $\text{DeleteEdge}_{\text{SSSP>0}}$ are necessary. With this information in hand it is possible to keep the total updating cost bounded.

In the previous two paragraphs, we have been speaking very roughly. In particular, because $\text{DeleteEdge}_{\text{SSSP>0}}$ as stated in Figure 1 actually performs the deletion of edge $v \longrightarrow w$ from graph *G* (see lines [2] and [33]), a few changes in $\text{DeleteEdge}_{\text{SSSP>0}}$ are necessary for it to be called multiple times in the manner suggested above.

There is also a more serious problem with using procedure $\text{DeleteEdge}_{\text{SSSP>0}}$ from Figure 1 in conjunction with the ideas outlined above. The problem is that $\text{DeleteEdge}_{\text{SSSP>0}}$ requires that shortest-path information be *explicitly* maintained for each sink *z* (*i.e.*, there would have to be *SP* sets for each sink *z*). For certain edge-modification operations, the amount of *SP* information that changes (for the entire collection of different sinks) is unbounded. In particular, when an edge $v \longrightarrow w$ is inserted with a length such that $length(v \longrightarrow w) = dist(v, w)$, there are no entries in the distance matrix that change value, and consequently

$$\begin{aligned} \| \delta \|_2 &= \| \text{MODIFIED} \|_2 + \sum_{u=1}^{|V(G)|} \| \text{AFFECTED}_u \|_2 \\ &= \| \text{MODIFIED} \|_2 . \end{aligned}$$

Such an insertion can introduce a new element in the *SP* set for each of the different sinks, and thus cause a change in *SP* information of size $\Omega(|V(G)|)$. Thus, using DeleteEdge$_{\text{SSSP>0}}$ from Figure 1 as a subroutine in DeleteEdge$_{\text{APSP>0}}$ would not yield a bounded incremental algorithm.

The way around these problems is to define a slightly different procedure, which we name DeleteUpdate, for use in DeleteEdge$_{\text{APSP>0}}$. Procedure DeleteUpdate is presented in Figure 5. DeleteUpdate is very similar to DeleteEdge$_{\text{SSSP>0}}$, but eliminates the two problems discussed above. DeleteUpdate does not delete any edges; the deletion of edge $v \longrightarrow w$ is performed in DeleteEdge$_{\text{APSP>0}}$ itself (see line [1] of Figure 6). In addition, DeleteUpdate does not need to update any *SP* information explicitly, because *SP*

```
procedure DeleteUpdate($G$, $v \longrightarrow w$, $z$)
declare
        $G$: a directed graph
        $v \longrightarrow w$: the edge that has been deleted from $G$
        $z$: the sink vertex of $G$
        WorkSet, AffectedVertices: sets of vertices
        $a, b, c, u, v, w, x, y$: vertices
        PriorityQueue: a heap of vertices
        $SP(a, b, c) \equiv (dist_G(a, c) = length_G(a \longrightarrow b) + dist_G(b, c)) \wedge (dist_G(a, c) \neq \infty)$
begin
[1]     AffectedVertices := $\emptyset$
[2]     if there does not exist any vertex $x \in Succ_G(v)$ such that $SP(v, x, z)$ then
[3]       /* Phase 1: Identify vertices in AFFECTED (the vertices whose shortest distance to $z$ has increased). */
[4]       /*          Set AffectedVertices equal to AFFECTED. */
[5]         WorkSet := { $v$ }
[6]         while WorkSet $\neq \emptyset$ do
[7]             Select and remove a vertex $u$ from WorkSet
[8]             Insert vertex $u$ into AffectedVertices
[9]             for each vertex $x \in Pred_G(u)$ such that $SP(x, u, z)$ do
[10]                if for all $y \in Succ_G(x)$ such that $SP(x, y, z)$, $y \in$ AffectedVertices then Insert $x$ into WorkSet fi
[11]            od
[12]        od
[13]      /* Phase 2: Determine new distances to $z$ for all vertices in AffectedVertices. */
[14]        PriorityQueue := $\emptyset$
[15]        for each vertex $a \in$ AffectedVertices do
[16]            $dist_G(a, z)$ := min ({ $length_G(a \longrightarrow b) + dist_G(b, z)$ |
                                  $a \longrightarrow b \in E(G)$ and $b \in (V(G) -$ AffectedVertices) } $\cup$ { $\infty$ })
[17]            if $dist_G(a, z) \neq \infty$ then InsertHeap(PriorityQueue, $a$, $dist_G(a, z)$) fi
[18]        od
[19]        while PriorityQueue $\neq \emptyset$ do
[20]            $a$ := FindAndDeleteMin(PriorityQueue)
[21]            for every vertex $c \in Pred_G(a)$ such that $length_G(c \longrightarrow a) + dist_G(a, z) < dist_G(c, z)$ do
[22]                $dist_G(c, z) := length_G(c \longrightarrow a) + dist_G(a, z)$
[23]                AdjustHeap( PriorityQueue, $c$, $dist_G(c, z)$)
[24]            od
[25]        od
[26]    fi
end
```

Figure 5. Procedure DeleteUpdate updates distances to vertex *z* after edge $v \longrightarrow w$ is deleted from *G*.

information is obtained when needed (in constant time) via the predicate $SP(a, b, c)$:

$$SP(a, b, c) \equiv (dist(a, c) = length(a \longrightarrow b) + dist(b, c)) \wedge (dist(a, c) \neq \infty).$$

Predicate $SP(a, b, c)$ answers the question "Is edge $a \longrightarrow b$ an SP edge when vertex c is the sink?". This check can be done in constant time.

The use of predicate $SP(a, b, c)$ makes it important that the test in line [10] be carefully implemented. Recall that Phase 1 is similar to a (reverse) topological order traversal in the SP DAG for sink z. We are interested in determining in line [10] if every successor of x in the SP DAG has already been "visited" and placed in AffectedVertices; if so, then x can be placed in AffectedVertices too. In procedure $\text{DeleteEdge}_{\text{SSSP}>0}$ we used the standard technique for performing a topological order traversal: a count was maintained at each vertex of the number of its successors (in the SP DAG) not yet placed in AffectedVertices; when the count for a vertex x fell to zero, it was placed in the WorkSet.

Since we cannot afford to maintain a similar count (across updates to the graph), we need to perform the check in line [10] differently. Note that the check in line [10] can be performed multiple times for the *same* vertex x. In fact, a vertex x can be checked $outdegree(x)$ times. If we examine all successors of vertex x each time, the cost of the repeated checks in line [10] for a *particular* vertex x can be quadratic in the number of successors it has. Instead, the same total cost can be made linear in $outdegree(x)$ by using the following strategy.

The first time vertex x is checked in line [10] we count the number of vertices y in $(Succ(x) - \text{AffectedVertices})$ that satisfy $SP(x,y,z)$. Whenever vertex x is subsequently checked in line [10] we decrement its count. We add x to the WorkSet when its count falls to zero.

Even this trick does not make the algorithm bounded in $\| \delta \|_1$. The reason is that the vertex x checked in line [10] is not necessarily a member of AFFECTED, but we are forced to examine all successors of x. However, even if the tested vertex x is not a member of AFFECTED it is guaranteed to be a predecessor of a member of AFFECTED. Consequently, the algorithm is bounded in $\| \delta \|_2$. In particular, the cost of Phase 1 is bounded by $O(\| \text{MODIFIED} \|_1 + \| \text{AFFECTED}_z \|_2)$; the cost of Phase 2 is bounded by $O(\| \text{AFFECTED}_z \|_1 + | \text{AFFECTED}_z | \log | \text{AFFECTED}_z |)$.

Procedure $\text{DeleteEdge}_{\text{APSP}>0}$ is given in Figure 6. Procedure $\text{DeleteEdge}_{\text{APSP}>0}$ actually maintains representations of *two* graphs: graph G itself and graph $\overline{G}$, the graph obtained by reversing the direction of every edge in G. This costs at most a factor of two in space and time. Thus, while the value $dist_G(u, v)$

```
procedure DeleteEdge_{APSP>0}(G, v ⟶ w)
declare
        G: a directed graph
        v ⟶ w: an edge to be deleted from G
        AffectedSinks, AffectedSources: sets of vertices
        v, w, x: vertices of G
begin
[1]     Remove edge v ⟶ w from E(G)
[2]     Remove edge w ⟶ v from E(Ḡ)
[3]     AffectedSinks := the set AffectedVertices from Phase 1 of DeleteUpdate(Ḡ, w ⟶ v, v)
[4]     AffectedSources := the set AffectedVertices from Phase 1 of DeleteUpdate(G, v ⟶ w, w)
[5]     for each vertex x ∈ AffectedSinks do DeleteUpdate(G, v ⟶ w, x) od
[6]     for each vertex x ∈ AffectedSources do DeleteUpdate(Ḡ, w ⟶ v, x) od
end
```

Figure 6. Procedure $\text{DeleteEdge}_{\text{APSP}>0}$ updates the solution to APSP>0 after edge $v \longrightarrow w$ is deleted from G.

stored at vertex u of graph G is the length of the shortest path from u to v in G, the value $dist_{\overline{G}}(u, v)$ is the length of the shortest path from v to u in G. Note that a single-*sink* problem in graph $\overline{G}$ is equivalent to a single-*source* problem in graph G. Thus, we will henceforth speak in terms of "solving single-source problems" synonymously with "solving single-sink problems in $\overline{G}$."

Both of these graphs are updated, as described earlier, by updating a collection of single-sink shortest-path problems on the corresponding graph. Exactly which single-sink problems need to be updated in G is determined by solving a distinguished single-sink problem in $\overline{G}$. The set AffectedVertices identified during this process indicates which single-sink problems must be updated in G. Similarly, the set AffectedVertices identified by solving a distinguished single-sink problem in G indicates which single-sink problems must be updated in $\overline{G}$. This duality is of crucial importance to achieving a bounded incremental update algorithm.

(1) The distinguished single-source problem is that of updating the distances from source-vertex v. This can be expressed as DeleteUpdate($\overline{G}$, $w \longrightarrow v$, v). The set AffectedVertices found during Phase 1 of this call indicates *exactly* which single-sink problems must be updated, for the following reasons:
   (i) For each vertex $x \in$ AffectedVertices found during Phase 1, there is at least one vertex (namely, vertex v) for which the length of the shortest path to x changed. That is, x is a sink for which some of the distances are out of date.
   (ii) Conversely, if z is any vertex for which there exists a vertex y such that the deletion of $v \longrightarrow w$ increases the length of the shortest path from y to z, then the old shortest path must have passed through $v \longrightarrow w$; consequently, the length of the shortest path from v to z must have changed as well. Thus, vertex z will be a member of AffectedVertices found during Phase 1 of the call on DeleteUpdate($\overline{G}$, $w \longrightarrow v$, v).

(2) By the dual argument, the set AffectedVertices found during Phase 1 of the call on DeleteUpdate(G, $v \longrightarrow w$, w) indicates *exactly* which single-source problems must be updated.

Consequently, the cost of DeleteEdge$_{\text{APSP>0}}$ is bounded by

$$O\left(\| \text{MODIFIED} \|_2 + \sum_{u=1}^{|V(G)|} \| \text{AFFECTED}_u \|_2 + \sum_{u=1}^{|V(G)|} | \text{AFFECTED}_u |_1 \log | \text{AFFECTED}_u |\right),$$

which in turn is bounded by $O(\| \delta \|_2 + | \delta | \log | \delta |)$.

3.2.2. Insertion of an Edge

We now present a bounded incremental algorithm for the problem of updating the solution to APSP>0 after an edge $v \longrightarrow w$ of length c is inserted into G. Though similar bounded algorithms have been previously proposed for this problem (see Rohnert [38], Even and Gazit [15], Lin and Chang [21], and Ausiello *et al.* [2]), we present the algorithm for the sake of completeness. Note that the algorithms described by Rohnert, Lin and Chang, and Ausiello *et al.* all maintain a shortest-path-tree data structure for each vertex, the maintenance of which can make the processing of an *edge-deletion* more expensive (and unbounded).

As in the case of edge deletion, we may obtain a bounded incremental algorithm for edge insertion as follows: compute AffectedSinks, the set of all vertices y for which there exists a vertex x such that the length of the shortest path from x to y has changed; for every vertex y in AffectedSinks, invoke the bounded incremental operation InsertEdge$_{\text{SSSP>0}}$ with y as the sink. The dual information maintained in $\overline{G}$ is updated in an identical fashion.

The algorithm InsertEdge$_{\text{APSP>0}}$ presented in Figure 8 carries out essentially the technique outlined above, but with one difference. It makes use of a considerably simplified form of the procedure InsertEdge$_{\text{SSSP>0}}$, which is given as procedure InsertUpdate in Figure 7. The simplifications incorporated in InsertUpdate are explained below.

```
procedure InsertUpdate($G$, $v \longrightarrow w$, $z$)
declare
        $G$: a directed graph
        $v \longrightarrow w$: the edge that has been inserted in $G$
        $z$: the sink vertex of $G$
        WorkSet: a set of edges
        VisitedVertices: a set of vertices
        $u, x, y$: vertices
        $SP(a, b, c) \equiv (dist_G(a, c) = length_G(a \longrightarrow b) + dist_G(b, c)) \wedge (dist_G(a, c) \neq \infty)$
begin
[1]     WorkSet := { $v \longrightarrow w$ }
[2]     VisitedVertices := { $v$ }
[3]     AffectedVertices := $\emptyset$
[4]     while WorkSet $\neq \emptyset$ do
[5]        Select and remove an edge $x \longrightarrow u$ from WorkSet
[6]        if $length_G(x \longrightarrow u) + dist_G(u,z) < dist_G(x,z)$ then
[7]           Insert $x$ into AffectedVertices
[8]           $dist_G(x,z) := length_G(x \longrightarrow u) + dist_G(u,z)$
[9]           for every vertex $y \in Pred_G(x)$ do
[10]             if $SP(y,x,v)$ and $y \notin$ VisitedVertices then
[11]                Insert $y \longrightarrow x$ into WorkSet
[12]                Insert $y$ into VisitedVertices
[13]             fi
[14]          od
[15]       fi
[16]    od
end
```

Figure 7. Procedure InsertUpdate updates distances to vertex z after edge $v \longrightarrow w$ is inserted into G.

Recall the description of InsertEdge$_{\text{SSSP}>0}$ given in Section 3.1.2. InsertEdge$_{\text{SSSP}>0}$ makes use of an adaptation of Dijkstra's algorithm to identify shortest paths to sink v and update distance information. However, in InsertUpdate, the DAG of all shortest paths to sink v is already available (albeit in an implicit form), and this information can be exploited to sidestep the use of a priority queue. (Note that the insertion of the edge $v \longrightarrow w$ cannot affect shortest paths to sink v, since the graph contains no cycles of negative length. Hence, the DAG of shortest paths to sink v undergoes no change during InsertEdge$_{\text{APSP}>0}$.) As explained in Section 3.2.1, the predicate $SP(a,b,v)$ can be used to determine, in constant time, if the edge $a \longrightarrow b$ is part of the DAG of shortest paths to sink v. This permits InsertUpdate to do a (partial) backward traversal of this DAG, visiting only affected vertices or their predecessors.

For instance, consider the edge $x \longrightarrow u$ selected in line [5] of Figure 7. Vertex x is the vertex to be visited next during the traversal described above. Except in the case when edge $x \longrightarrow u$ is $v \longrightarrow w$, vertex u is an affected vertex and is the successor of x in a shortest path from x to v. The test in line [6] determines if x itself is an affected vertex. If it is, its distance information is updated, and its predecessors in the shortest-path DAG to sink v are added to the workset for subsequent processing, unless they have already been visited. The purpose of the set VisitedVertices is to keep track of all the vertices visited in order to avoid visiting any vertex more than once. For reasons to be given shortly, InsertUpdate simultaneously computes AffectedVertices, the set of all vertices the length of whose shortest path to vertex z changes.

We now justify the method used in InsertEdge$_{\text{APSP}>0}$ to determine AffectedSinks, the set of all vertices y for which there exists a vertex x such that the length of the shortest path from x to y has changed. This set is the set of sinks for which InsertEdge$_{\text{APSP}>0}$ must invoke InsertUpdate. Assume that x and y are

```
procedure InsertEdge_{APSP>0}($G$, $v \longrightarrow w$, $c$)
declare
        $G$: a directed graph
        $v \longrightarrow w$: an edge to be inserted in $G$
        $c$: a positive real number indicating the length of edge $v \longrightarrow w$
        AffectedSinks, AffectedSources: sets of vertices
        $v$, $w$, $x$: vertices of $G$
begin
[1]     Insert edge $v \longrightarrow w$ into $E(G)$
[2]     Insert edge $w \longrightarrow v$ into $E(\overline{G})$
[3]     $length_G(v \longrightarrow w) := c$
[4]     $length_{\overline{G}}(w \longrightarrow v) := c$
[5]     AffectedSinks := the set AffectedVertices from InsertUpdate($\overline{G}$, $w \longrightarrow v$, $v$)
[6]     AffectedSources := the set AffectedVertices from InsertUpdate($G$, $v \longrightarrow w$, $w$)
[7]     for each vertex $x \in$ AffectedSinks do InsertUpdate($G$, $v \longrightarrow w$, $x$) od
[8]     for each vertex $x \in$ AffectedSources do InsertUpdate($\overline{G}$, $w \longrightarrow v$, $x$) od
end
```

Figure 8. Procedure InsertEdge$_{\text{APSP>0}}$ updates the solution to APSP>0 after edge $v \longrightarrow w$ of length c is inserted in G.

vertices such that the length of the shortest path from x to y changes following the insertion of edge $v \longrightarrow w$. Then, the new shortest path from x to y must pass through the edge $v \longrightarrow w$. Obviously, the length of the shortest path from v to y must have changed as well. Hence, AffectedSinks is the set $\{ y \mid$ the length of the shortest path from v to y changes following the insertion of edge $v \longrightarrow w \}$. This set is precisely the set of all affected vertices for the single-source shortest-path problem with v as the source, *i.e.* the set AffectedVertices computed by the call InsertUpdate($\overline{G}$,$w \longrightarrow v$,v). This is how InsertEdge$_{\text{APSP>0}}$ determines the set AffectedSinks (see line [5] of Figure 8); InsertUpdate is then invoked repeatedly, once for each member of AffectedSinks. The update to graph $\overline{G}$ is performed in an analogous fashion.

We now consider the time complexity of InsertEdge$_{\text{APSP>0}}$. Note that for every vertex $x \in$ AffectedSinks, any vertex examined by InsertUpdate(G,$v \longrightarrow w$,x) is in N(AFFECTED$_x$). InsertUpdate does essentially a simple traversal of the graph $<N(\text{AFFECTED}_x)>$, in time $O(\| \text{AFFECTED}_x \|)$. Thus, the total running time of line [7] in procedure InsertEdge$_{\text{APSP>0}}$ is $O(\| \delta \|_1)$. Similarly, line [8] takes time $O(\| \delta \|_1)$. Line [5] takes time $O(\| \text{AFFECTED}_v \|_{1,\overline{G}})$; line [6] takes time $O(\| \text{AFFECTED}_w \|_{1,G})$. Thus, the total running time of procedure InsertEdge$_{\text{APSP>0}}$ is $O(\| \delta \|_1)$.

3.3. The Dynamic Circuit-Annotation Problem

A *circuit* is a DAG in which every vertex u is associated with a function F_u. The output value to be computed at any vertex u is obtained by applying function F_u to the values computed at the predecessors of vertex u. The circuit-annotation problem, also known as the circuit-value problem, is to compute the output value associated with each vertex. Alpern *et al.* show that the incremental circuit-annotation problem has a lower bound of $\Omega(2^{\|\delta\|})$ under a certain model of incremental computation [1]. In this section we develop an algorithm for the incremental circuit-annotation problem that runs in time $O(2^{\|\delta\|})$, under the assumption that the evaluation of each function F_u takes unit time[3]. Previous to our work, no bounded

[3]In general, it is not true that each function F_u in a circuit can be computed in unit time. For instance, it might be necessary to look at the values of all the predecessors of vertex u in order to compute the value at u. In this case, it might be more reasonable to assume that the cost of computation of F_u is proportional to the indegree of vertex u. A variant of the incremental algorithm presented in this section runs in $O(\| \delta \|_2 \cdot 2^{\|\delta\|_2})$ time under this assumption. We do not describe the variant here due to space limitations. See [29]

algorithm for the dynamic circuit-annotation problem was known.

Consider a circuit whose vertices are annotated with (output) values. The value annotating vertex u will be denoted by *u.value*. Vertex u is said to be *consistent* if its value equals function F_u applied to the values associated with its predecessor vertices. The circuit is said to be *correctly annotated* if each vertex in the circuit is consistent. A vertex is said to be *correct* if its value is the one it would have in a correct annotation of the circuit. Note that a consistent vertex might be incorrect (but only if at least one of its predecessors is incorrect). A change to the circuit consists of the insertion or deletion of a vertex u, or the modification of the function F_u, or the insertion or deletion of an edge $v \longrightarrow u$. Obviously, if the initial circuit was correctly annotated, then at most vertex u could be inconsistent in the modified circuit. Consequently the dynamic circuit-annotation problem is: given an annotated circuit G, and a vertex u in G such that every vertex in G except possibly u is consistent, compute the correct annotation of G. The vertex u is the *modified* vertex.

Proposition 3. *The dynamic circuit-annotation problem has a bounded incremental algorithm, which processes a change* δ *in time* $O(2^{||\delta||})$.

The algorithm outlined in this section is a change-propagation algorithm. In a change-propagation algorithm, the output values of certain *potentially affected* vertices are recomputed. If the new value at any vertex v is different from its original value (*i.e.*, the value before the update began), v's successor vertices are deemed potentially affected. In order to avoid extra computation it is necessary to visit potentially affected vertices in a topological-sort order. This requires maintaining information that assists in visiting the vertices in a topological-sort order. This is the approach taken by Alpern *et al.*[1]. A DAG is said to be *correctly prioritized* if every vertex u in the DAG is assigned a priority, denoted by $priority(u)$, such that if there is a path in the DAG from vertex u to vertex v then $priority(u) < priority(v)$. Alpern *et al.* outline an algorithm for the problem of maintaining a *correct prioritization* of a circuit in the presence of modifications. They utilize the priorities in propagating changes in the circuit in a topological-sort order. This, however, leads to an unbounded algorithm for the dynamic circuit-annotation problem. This is because maintaining a topological-sort ordering or priority ordering of the DAG can require time unbounded in terms of $||\delta||$, since the topological ordering of the vertices might be greatly changed following modification δ, yet none of the output values might have changed. Thus, we cannot afford to maintain priorities or a topological ordering of the vertices of the circuit if we desire a bounded algorithm for the dynamic circuit-annotation problem.

The change-propagation algorithm we describe below does not maintain any topological ordering of the DAG (and hence, in general, *does* perform extra computations that will be undone later on). Instead, the algorithm makes use of a *relative topological-sort ordering* of a set of vertices. Let H denote the subgraph of G induced by a set of vertices S. Any topological sorting of H is said to yield a *relative topological-sort ordering* for S. Note in particular that if a vertex u topologically precedes vertex v in G and all paths in G from u to v pass through some vertex not in S, then u need not come before v in a relative topological-sort ordering for S. This is important because, in general, it is not possible to determine an actual topological-sort ordering of of a set S (*i.e.*, an ordering that accounts for all paths in G) in time bounded by a function of $|S|$ (or even $||S||_i$ for any fixed value of i). In contrast, under the assumption that each vertex has a bounded number of successors, it is possible to determine a relative topological-sort ordering of the vertices of S in time $O(|S|)$.

---

for details.

We now outline an algorithm, called *UpdateCircuit*, that processes changes to a "binary" circuit in time $O(2^{||\delta||})$. (A binary circuit is one in which every vertex has outdegree less than or equal to 2.) The algorithm is described in Figure 9, and it works as follows. The algorithm initializes the set WorkSet to consist of the modified vertex, which is the only vertex in the circuit that can be inconsistent. In each iteration of the loop in lines [3]-[12] the values of all the vertices in WorkSet are recomputed in a relative topological-sort ordering. The set of all vertices in WorkSet that have a value different from their original value is identified in line [5]. These vertices are said to be *apparently affected*—some of these vertices may not be affected but just have a wrong value temporarily assigned to them. The set of all successors of the apparently affected vertices, the *potentially affected* vertices, is identified in line [6]. The algorithm halts if all the potentially affected vertices are already in WorkSet. Otherwise, the potentially affected vertices are added to WorkSet and the algorithm iterates through this process again.

Proposition 4. *Procedure UpdateCircuit computes a correct annotation of G.*

Proof. Consider the circuit as annotated when the procedure terminates. We show that every vertex in the circuit is correctly annotated by induction on the vertices *v* of *G* in "topological-sort order": we show for every vertex *v* in *G* that the inductive hypothesis that every predecessor of *v* in *G* is correct implies that *v* is itself correct.

Let $\overline{\text{WorkSet}}$ denote the final value of WorkSet. First consider the case that *v* is in $\overline{\text{WorkSet}}$. Since the values for vertices in $\overline{\text{WorkSet}}$ have been computed in a relative topological-sort order, it follows that every vertex in $\overline{\text{WorkSet}}$ is consistent. (Whenever *v.value* is recomputed, *v* becomes consistent. It can subsequently become inconsistent only if the value of some predecessor of *v* changes.) It follows that vertex *v* is also correct since, according to the inductive hypothesis, all the predecessors of *v* are correct.

```
procedure UpdateCircuit (G, u)
declare
        G : an annotated circuit
        u : the modified vertex in G
        WorkSet, ApparentlyAffected, PotentiallyAffected : sets of vertices
        v: a vertex
preconditions
        Every vertex in V(G) except possibly u is consistent
begin
[1]     WorkSet := { u }
[2]     u.originalValue := u.value
[3]     loop
[4]        for every vertex v ∈ WorkSet in relative topological-sort order do recompute v.value od
[5]        ApparentlyAffected := { v ∈ WorkSet : v.value ≠ v.originalValue }
[6]        PotentiallyAffected := Succ (ApparentlyAffected)
[7]        if PotentiallyAffected ⊆ WorkSet then exit loop fi
[8]        for every vertex v ∈ (PotentiallyAffected − WorkSet) do
[9]           Insert v into WorkSet
[10]          v.originalValue := v.value
[11]       od
[12]    end loop
end
postconditions
        Every vertex in G is consistent
```

Figure 9. An algorithm for the dynamic circuit-annotation problem.

Now consider the case that v is not in $\overline{\text{WorkSet}}$. Note that the following condition holds true when the procedure terminates: if w and v vertices such that $w \in \overline{\text{WorkSet}}$, $v \notin \overline{\text{WorkSet}}$, $w \longrightarrow v \in E(G)$, then $w.value = w.originalValue$. Hence, any predecessor w of v that is in $\overline{\text{WorkSet}}$ has the same value as it did originally. Since only the values of vertices in $\overline{\text{WorkSet}}$ could have changed, any predecessor of v that is not in $\overline{\text{WorkSet}}$ has the same value as it did initially. Hence, v and all of its predecessors have the same values as they did before the update. Since v was initially consistent (from the precondition of the procedure), it must still be consistent and, hence, correct. It follows that *UpdateCircuit* computes a correct annotation of the circuit.

Proposition 5. *Procedure UpdateCircuit computes the correct annotation of a binary circuit G in time* $O(2^{|\text{AFFECTED}|})$.

Proof. The proof that the computed annotation is correct follows from Proposition 4. The proof of the time complexity follows.

We first show that the algorithm adds at least one affected vertex to WorkSet in each of the iterations except possibly the last two. Assume that after the execution of line [7] in the i-th iteration of the outer loop (lines [3]-[12]), every vertex in PotentiallyAffected−WorkSet is an unaffected vertex. In other words, all the vertices that are added to WorkSet in the i-th iteration of the outer loop are assumed to be unaffected vertices. Then, we can show that the circuit must be correctly annotated at this point using induction on the vertices in a topological-sort order: we show for every vertex v in G that the inductive hypothesis that every predecessor of v in G is correct implies that v is itself correct.

First consider the case that v is in WorkSet. Since the values for vertices in WorkSet have been computed in a relative topological-sort order, it follows that every vertex in WorkSet is consistent. It follows that vertex v is also correct.

Now consider the case that v is in PotentiallyAffected−WorkSet. Thus, v is one of the vertices that is added to WorkSet in the i-th iteration. Hence, v is an unaffected vertex, according to our hypothesis, and is correct.

Let v be in neither PotentiallyAffected nor WorkSet. Then every predecessor of v must have the same value as it did initially. (Otherwise, v would be in PotentiallyAffected.) Since v has the same value as it did initially, and since v was initially consistent, it follows that v is still consistent. It follows that v is correct.

Thus, the circuit has a correct annotation at the end of the i-th iteration. Hence, the subsequent iteration will not change any of the output values. (Note that re-evaluation of a consistent vertex does not change its value.) Consequently, the algorithm halts after the $i+1$-th iteration.

It follows from the above argument that the algorithm makes at most $|\text{AFFECTED}|+1$ iterations.

Because every vertex in the circuit has outdegree at most 2, at most 2^i new vertices can be added to WorkSet during the i-th iteration. Hence, at the beginning of the i-th iteration, $|\text{WorkSet}| \le \sum_{j=0}^{i-1} 2^j = (2^i - 1)$. The i-th iteration itself takes time $O(2^i)$. The whole algorithm takes time $O(\sum_{i=1}^{|\text{AFFECTED}+1|} 2^i) = O(2^{|AFFECTED|})$. □

Aside. There are obvious improvements that can be made to the above algorithm. WorkSet undergoes incremental changes during every iteration, and the various computations performed during each iteration may be performed in an incremental fashion. Thus, for instance, there is no need to recompute the value for every vertex in WorkSet during each iteration. Such changes improve the average-case performance, but the worst-case complexity would still be exponential in $\|\delta\|$. Experimental results show that with such improvements, the above algorithm is actually a practical one, at least in some contexts such as language-sensitive editors. See [29]. *End Aside*.

Note that even if the circuit G is not binary, *UpdateCircuit* will compute the correct annotation of G. However, it may not do so in time bounded by any function of $\|\delta\|$. The reason is that in procedure *UpdateCircuit*, an unaffected vertex z, which by definition is initially correct, may be given an incorrect value at some intermediate iteration i. Although, z's correct value will ultimately be restored by the time *UpdateCircuit* terminates, z's successors are part of the WorkSet at the end of iteration i; because z is not affected, this may cause $|\text{WorkSet}|$ to be unbounded in $\|\delta\|$.

We can, however, use procedure *UpdateCircuit* to obtain an $O(2^{\|\delta\|})$ algorithm for general circuits as follows. Given a circuit G, we can construct a binary circuit G^* that is equivalent to G in some sense, as follows. Let u be a vertex in G with k successors $v_1, \cdots, v_k$ where $k > 2$. Replace u by $k-1$ vertices $u_1, \cdots, u_{k-1}$ each of out-degree 2. Vertex u_1 has the same function and the same set of predecessors as vertex u, and two successors v_1 and u_2. For $1 < i \le k-1$, vertex u_i has a single predecessor u_{i-1}, and is associated with the identity function. Each of these vertices except u_{k-1} has two successors v_i and u_{i+1}. u_{k-1} has two successors v_{k-1} and v_k. G^* is obtained from G by splitting all vertices of G with outdegree greater than 2 in this fashion.

It is not really necessary to construct the circuit G^*. We can effectively simulate the action of *UpdateCircuit* on G^*, given just G. This leads to an $O(2^{\|\delta\|})$ algorithm for general circuits.

4. LOWER-BOUND RESULTS: PROBLEMS THAT ARE NON-INCREMENTAL FOR LOCALLY PERSISTENT ALGORITHMS

The class of *locally persistent* algorithms was introduced by Alpern *et al*. in [1]. What follows is their description of this class of algorithms, paraphrased to be applicable to general graph problems.

A *locally persistent* algorithm may make use of a block of storage for each vertex of the graph. (This may be directly generalized to permit storage blocks to be associated with edges, too.) The storage block for vertex u will include pointers to (the blocks of storage for) the predecessor and successor vertices of u. The storage block for u will contain the output value for u. The block may also contain an arbitrary amount of auxiliary information, but no auxiliary pointers (to vertices, *i.e.*, their storage blocks). No global auxiliary information is maintained in between successive modifications to the graph: whatever information persists between calls on the algorithm is distributed among the storage blocks for the vertices. An input change is represented by a pointer to the vertex or edge modified. A locally persistent algorithm begins with the representation of a change and follows pointers. The choice of which pointer to follow next may depend (in any deterministic way) on the information at the storage blocks visited so far. For example, a locally persistent algorithm may make use of worklists or queues of vertices adjacent to those vertices that have already been visited. The auxiliary information at a visited storage block may be updated (again in any way that depends deterministically on the information at the visited storage blocks).

In summary, these algorithms have two chief characteristics. First, any auxiliary information used by the algorithm is associated with an edge or a vertex of the graph—no information is maintained globally. Second, the algorithm starts an update from the vertices or edges that have been modified and traverses the graph using only the edges of the graph. In essence, the auxiliary information at a vertex or edge cannot be used to access non-adjacent vertices and edges.

In this section, we show that the problem of graph reachability is unbounded for the class of locally persistent algorithms (*i.e.*, the problem has no bounded locally persistent incremental algorithm). We also show, using reduction from reachability, that two large classes of problems—the *closed-semiring path problems* and the *meet-semilattice data-flow analysis problems*—are unbounded for the class of locally persistent algorithms. These lower bound results also hold with respect to a more powerful model of computation (a restricted pointer machine model), but we present only the proof for the class of locally persistent algorithms due to space considerations. The more general proof may be found in [29]

Throughout the section, unless explicitly noted otherwise, the term “unbounded” is shorthand for “unbounded for the class of locally persistent algorithms.”

4.1. The Single-Source Reachability Problem is Unbounded

Definition 6. (*The single-source reachability problem: SS-REACHABILITY.*) Given a directed graph G with a distinguished vertex s (the *source*), determine for each vertex u whether u is reachable from s (*i.e.*, whether there is a path in the graph G from s to u).

Proposition 7. *SS-REACHABILITY is unbounded for locally persistent algorithms.*

Proof. The lower bound is established by constructing a graph G and two “trivial” changes δ_1 and δ_2 in the graph that are far “apart” such that there is some “interaction” between the two changes. The changes are trivial in that both $G+\delta_1$ and $G+\delta_2$ have the same solution as G. The changes interact in that $G+\delta_1+\delta_2$ has a different solution from G. The changes are far apart in a sense that we now define.

Given vertices u and v in an undirected graph G, let $d_G(u,v)$, the *distance* between u and v, denote the length of (*i.e.*, the number of edges in) the shortest path between u and v in G. If U and W are two sets of vertices (or two subgraphs) of G, then $d_G(U,W)$ is defined to be the shortest distance between some vertex of U and some vertex of W. Thus, $d_G(U,W) = \min \{ d_G(u,v) \mid u \in U, v \in W \}$, if U and W are sets of vertices. Similarly, if H and F are subgraphs of G, $d_G(H,F) = \min \{ d_G(u,v) \mid u \in V(H), v \in V(F) \}$.

Given a (directed) graph G, we will denote the underlying undirected graph by $\underline{G}$. Assume that δ_1 and δ_2 are two modifications that convert graph G to graphs H_1 and H_2, respectively. Let J denote the union of the graphs $\underline{G}$, $\underline{H_1}$ and $\underline{H_2}$. (The union of two graphs $\underline{G}$ and $\underline{H}$ is the graph $(V(\underline{G}) \cup V(\underline{H}),\ E(\underline{G}) \cup E(\underline{H}))$.) The distance $d_G(\delta_1,\delta_2)$ between the two modifications δ_1 and δ_2 to graph G is defined to be $d_J(\mathrm{MODIFIED}_{G,\delta_1}, \mathrm{MODIFIED}_{G,\delta_2})$.

Consider the graph G shown in Figure 10. Let δ_1 denote the deletion of the edge $u \longrightarrow x$ and let δ_2 denote the deletion of the edge $v \longrightarrow y$. Let H_1 and H_2 denote the graphs $G+\delta_1$ and $G+\delta_2$, respectively. Obviously, none of the vertices in H_1 or H_2 are affected, and thus, for any fixed value of i, $\| \delta_1 \|_{i,G} = \| \delta_2 \|_{i,G} = O(1)$. The proof involves showing that a locally persistent algorithm cannot process both the change δ_1 to G and the change δ_2 to G in constant time (*i.e.*, time independent of the size of graph G or the length of the dotted paths indicated in the figure).

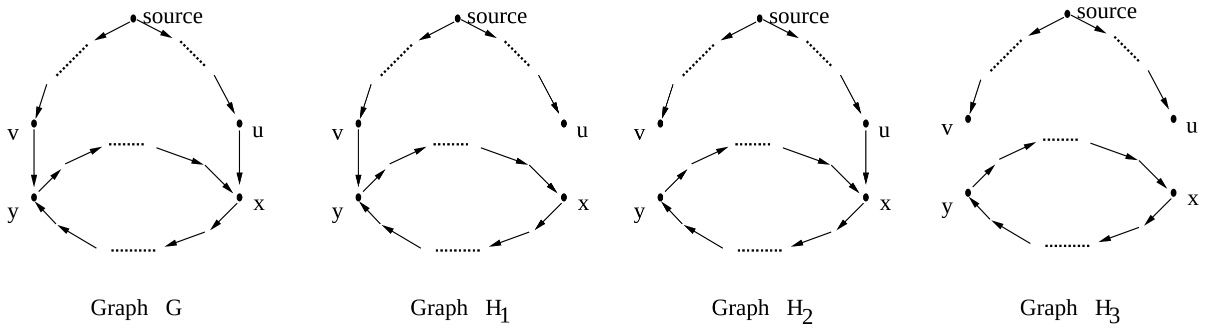


Figure 10. Graphs used in the proof that SS-REACHABILITY is unbounded for locally persistent algorithms.

Consider any locally persistent incremental algorithm for SS-REACHABILITY. Let $Trace(G', \delta')$ denote the sequence of steps executed by the algorithm in processing some change δ' to some graph G'. Consider the following two instances: the application of modification δ_2 to graph G and the application of modification δ_2 to graph H_1. Obviously, the update procedure must behave differently in these two cases, and $Trace(G, \delta_2)$ must be different from $Trace(H_1, \delta_2)$ (because many vertices of $H_3 = H_1 + \delta_2$ are affected, whereas no vertex in $G + \delta_2$ is affected). Since a locally persistent algorithm makes use of no global storage, this can happen only if both $Trace(G, \delta_2)$ and $Trace(H_1, \delta_2)$ include a visit to some vertex w that contains different information in the graphs G and H_1. But H_1 was obtained from G by making change δ_1. Hence, the information at vertex w must have been changed during the updating that followed the application of change δ_1 to G. It follows that $Trace(G, \delta_1)$ must contain a visit to vertex w. A characteristic of locally persistent algorithms is that if a vertex w is visited during the updating that follows the application of change δ' to graph G', then every vertex in some path in graph $\underline{G'}$ from a modified vertex to w must have been visited. Consequently, $Trace(G, \delta_1)$ and $Trace(G, \delta_2)$, between them, include visits to every vertex on some path from x to y in $\underline{G}$. Hence, the time taken for processing change δ_1 to G plus the time taken for processing change δ_2 to $\overline{G}$ must be $\Omega(d_G(\delta_1, \delta_2))$. But, $d_G(\delta_1, \delta_2)$ can be unbounded, *i.e.*, $\Theta(|G|)$. Hence, any locally persistent incremental algorithm for SS-REACHABILITY must be unbounded. $\square$

In the following sections we show that various path problems in graphs and various data-flow analysis problems are all unbounded by reducing the reachability problem to these problems. The reductions utilize a “homomorphic embedding” of the reachability problem into these other problems.

4.2. Unbounded Path Problems

We now show that several other graph problems are also unbounded. These graph problems are best described using the *closed-semiring* framework.

Definition 8. A *closed semiring* is a system $(S, \oplus, \otimes, \overline{0}, \overline{1})$ consisting of a set S, two binary operations $\oplus$ and $\otimes$ on S, and two elements $\overline{0}$ and $\overline{1}$ of S, satisfying the following axioms:

(1) $(S, \oplus, \overline{0})$ is a meet-semilattice with greatest element $\overline{0}$. (Thus, $\oplus$ is a commutative, associative, idempotent operator with identity element $\overline{0}$. The meet operator will also be referred to as the summary operator.) Further, the meet (summary) of any countably infinite set of elements $\{ a_i \mid i \in N \}$ exists and will be denoted by $\bigoplus_{i \in N} a_i$.

(2) $(S, \otimes, \overline{1})$ is a monoid. (Thus, $\otimes$ is an associative operator with identity $\overline{1}$.)

(3) $\otimes$ distributes over finite and countably infinite meets: $(\bigoplus_i a_i) \otimes (\bigoplus_j b_j) = \bigoplus_{i,j} (a_i \otimes b_j)$.

(4) $a \otimes \overline{0} = \overline{0}$.

A unary operator $*$, called *closure*, of a closed semiring $(S, \oplus, \otimes, \overline{0}, \overline{1})$ is defined as follows:

$$a* =_{def} \bigoplus_{i=0}^{\infty} a^i$$

where $a^0 = \overline{1}$ and $a^{i+1} = a^i \otimes a$.

Different path problems in directed graphs are captured by different closed semirings. An instance of a given path problem involves a directed graph $G = (V, E)$ and an edge-labeling function that associates a value from S with each $e \in E$.

Consider a directed graph G, and a label function l that maps each edge of G to an element of the set S. The function l can be extended to map paths in G to elements of S as follows. The *label* of a path $p = [e_1, e_2, \cdots, e_n]$ is defined by $l(p) = l(e_1) \otimes l(e_2) \otimes \cdots \otimes l(e_n)$. If v, w are two vertices in the graph, then $C(v, w)$ is defined to be the meet (summary) over all paths p from v to w of $l(p)$:

$$C(v,w) = \bigoplus_{v \longrightarrow_p w} l(p).$$

The closed-semiring framework for path problems captures both "all-pairs" problems and "single-source" problems. In an all-pairs problem, the goal is to compute $C(v,w)$ for all pairs of vertices $v, w \in V(G)$. In a single-source problem, the goal is to compute only the values $C(s,w)$ where s is the distinguished source vertex. In all these problems, (unit-time) operations implementing the operators $\oplus$, $\otimes$, and * are assumed to be available. More formally, let **R** = $(S, \oplus, \otimes, \overline{0}, \overline{1})$ be a specific closed semiring. The SS-**R** problem is defined as follows.

Definition 9. Given a directed graph $G = (V, E)$, a vertex s in V, and an edge-labeling function $l : E \longrightarrow S$, the *SS*-**R** *problem* is to compute $C(s,w)$ for every vertex w in V. We say that (G, s, l) is an *instance* of the SS-**R** problem.

In the dynamic version of the SS-**R** problem that we consider, the source vertex s is assumed to be fixed.

For example, let **R** be the closed-semiring $(\mathrm{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$. Then, SS-**R** is nothing other than the single-source shortest-path problem with non-negative edge lengths.

In this section we show that for any closed semiring **R**, the SS-**R** problem is unbounded. We first show that the SS-**R** problem is "at least as difficult as" the SS-REACHABILITY problem, even for incremental algorithms, by "reducing" the SS-REACHABILITY problem to the SS-**R** problem, and conclude that the SS-**R** problem is unbounded.

However, some caution needs to be exercised in making inferences about the unboundedness of a problem via a reduction argument. If a problem P is unbounded and can be reduced to a problem Q in the conventional sense, it does not necessarily follow that the problem Q is unbounded. For instance, consider any unbounded problem P of computing some value $S(u)$ for each vertex u of the graph. Consider the (intuitively) "more difficult" problem Q of computing $S(u)$ *and* $T(u)$ for each vertex u of the graph, where $T(u)$ is defined such that it changes whenever the input changes. For example, let $T(u)$ be the sum of the number of vertices and the number of edges in the graph. If each input change consists of the addition or deletion of a vertex or an edge, then by definition, whenever the input changes *every* vertex is affected. Consequently, any update algorithm is a bounded algorithm, and Q is a bounded problem.

Showing that a problem Q is unbounded by reducing an unbounded problem P to Q involves the following obligations: (1) We must show how every instance of problem P (*i.e.*, the input) can be transformed into an instance of problem Q, and how the solution for this transformed problem instance can be translated back into a solution for the original problem instance. (2) We must show how any change δ_P to the original problem instance can be transformed into a corresponding change δ_Q in the target problem instance, and, similarly, how the change in the solution to the target problem instance can be transformed into the corresponding change in the solution to the original problem instance. (3) We must show that the time taken for the transformations referred to in (2) is *bounded* by some function of $\|\delta_P\|$. (4) We must show that $\|\delta_Q\|$ is also bounded by some function of $\|\delta_P\|$. (5) Finally, since we are dealing with the notion of unboundedness relative to the class of locally persistent algorithms, we must show that the transformation algorithms referred to in (2) are locally persistent.

Proposition 10. *Let* **R** = $(S, \oplus, \otimes, \overline{0}, \overline{1})$ *be an arbitrary closed semiring. The SS-***R** *problem is unbounded for the class of locally persistent algorithms.*

Proof. Given an instance of a single-source reachability problem (G, s), there is a linear-time reduction to an instance of SS-**R** given by $(G, s, \lambda e.\overline{1})$. In the target problem instance, the summary value at v, $C(s,v)$ is $\overline{1}$ if v is reachable from s, and $\overline{0}$ otherwise.

It is obvious that all the requirements laid down above for reduction among dynamic problems are met by the above reduction. Therefore, SS-**R** is an unbounded problem. □

It follows from the above proposition that SSSP≥0 is an unbounded problem. However, as we saw in Section 3.1, the very similar problem SSSP>0 has a bounded locally persistent incremental algorithm. This illustrates that only certain input instances may be the reason why a problem is unbounded. For example, graphs with 0-length cycles are what causes SSSP≥0 to be unbounded. If the problematic input instances are unrealistic in a given application, it would be appropriate to consider a suitably restricted version of the problem that does not deal with these difficult instances.

4.3. Non-Incremental Data-Flow Analysis Problems

In this section we show that all non-trivial meet-semilattice data-flow analysis problems are unbounded. Data-flow analysis problems are often cast in the following framework. The program gives rise to a *flow graph* G with a distinguished *entry* vertex s. Without loss of generality, s may be assumed to have no incoming edges. The problem requires the computation of some information $S(u)$ for each vertex u in the flow graph. The values $S(u)$ are elements of a meet semilattice L; a (monotonic) function $M(e){:}L \longrightarrow L$ is associated with every edge e in the flow graph; and a constant $c \in L$ is associated with the vertex s. The desired solution $S(u)$ is the maximal fixed point of the following collection of equations:

$$S(s) = c$$
$$S(u) = \bigsqcap_{v \longrightarrow u \in E(G)} M(v \longrightarrow u)(S(v)), \quad \text{for } u \neq s.$$

Each semilattice L and constant $c \in L$, often the greatest or least element of the semilattice, determines a data-flow analysis problem, which we call the (L,c)-DFA problem. An input instance of the problem consists of a graph G and a mapping M from the edges of G to $L \longrightarrow L$.

We now show that an arbitrary meet-semilattice data-flow analysis problem P is unbounded by reducing SS-REACHABILITY to P.

Proposition 11. *Let L be a meet-semilattice, and let $c \in L$. Then, the problem (L,c)-DFA is unbounded for the class of locally persistent algorithms.*

Proof. Let f be a function from L to L such that $f(c) \neq \top$. Given an instance $((V,E),s)$ of the single-source reachability problem we can construct a corresponding instance $((V \cup \{t\}, E \cup \{(t \longrightarrow s)\}), t, M)$ of problem P where,

$$M(e) = f \qquad \text{if } e = t \longrightarrow s$$
$$M(e) = \lambda x.x \quad \text{if } e \neq t \longrightarrow s.$$

The solution of this problem instance is given by: $S(t) = c$; if $u \neq t$, then $S(u)$ is $f(c)$ if u is reachable from s, and $\top$ otherwise. It follows from the unboundedness of SS-REACHABILITY that P is unbounded. □

The interpretation of the above result is that any locally persistent incremental algorithm for problem P is an unbounded algorithm. This does not by itself imply that the data-flow analysis problem P that arises in practice is an unbounded one for locally persistent algorithms (in other words, if there is some flexibility in defining the class of valid input instances for problem P). The above reduction shows that some “difficult” input instances cannot be handled in time bounded by a function of $\|\delta\|$. However, these input instances may be unrealistic input instances in the context of the data-flow analysis problem under consideration. We now argue that, in fact, this is not the case.

The first possible restriction on input instances relates to the flow graph. Ordinarily, frameworks for batch data-flow analysis problems impose the assumption that all vertices in a flow graph be reachable from the graph’s start vertex. Some data-flow analysis algorithms also assume that the data-flow graph is a reducible one. With either of these restrictions on input instances, the above reduction of SS-REACHABILITY to problem P is no longer valid. However, we follow Marlowe [23], who argued that these assumptions should be dropped for studies of incremental data-flow analysis (see Section 3.3.1 of [23]).

The second possible restriction on input instances relates to the mapping M. Is it possible that realistic flow-graphs will never have a labeling corresponding to the "difficult" input instances shown to exist above? We argue below that this is not so.

The reduction above associated every edge with either the identity function or a function f such that $f(c) \neq \top$. The identity function is not an unrealistic label for an edge. (A **skip** statement, or more generally, any statement that modifies the state in a way that is irrelevant to the information being computed by the data-flow analysis problem P is usually associated with the identity function.) As for the function f, we now show that every non-trivial input instance must have an edge labeled by a function g such that $g(c) \neq \top$. Consider any input instance (G, s, M) such that $M(e)(c) = \top$ for every edge $e \in E(G)$. Since $M(e)$ must be monotonic, $M(e)(\top)$ must also equal $\top$. Then, the input instance (G, s, M) has the trivial solution given by:

$$S(s) = c$$
$$S(u) = \top \quad \text{for } u \neq s.$$

Hence, the edge-labeling M from the reduction used in the proof of Proposition 11 is, in fact, realistic.

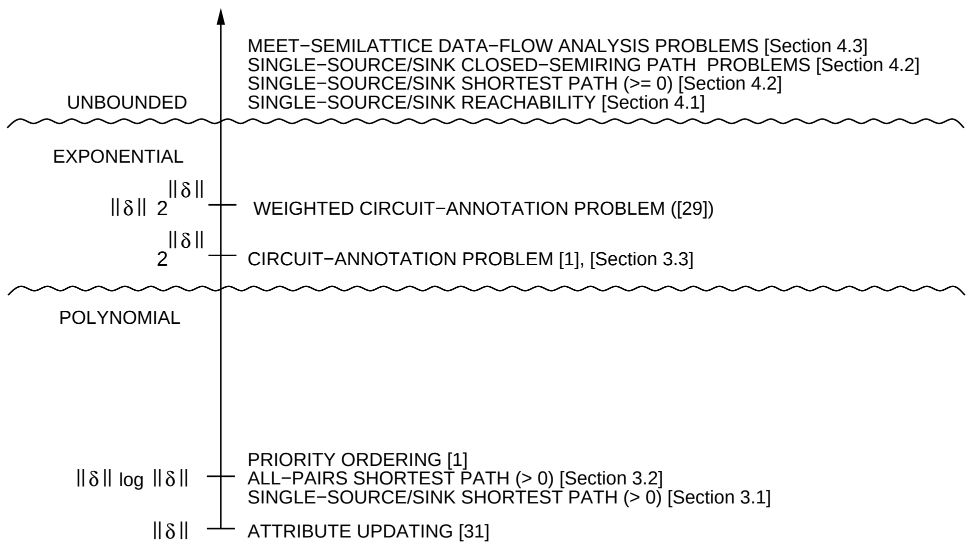
In conclusion, note that the reduction used in the proof of Proposition 11 is independent of the class of incremental algorithms proposed (*i.e.*, locally persistent or otherwise). That is, the incremental version of every data-flow analysis problem is at least as hard as the dynamic single-source reachability problem. In other words, for a class of algorithms to have members that are bounded for any data-flow analysis problem, there must be an algorithm of the class that solves the single-source reachability problem in a bounded fashion.

5. A Computational-Complexity Hierarchy for Dynamic Graph Problems

The results from Sections 3 and 4, together with some previously known results, allow us to begin to understand the structure of the complexity hierarchy that exists when dynamic problems are classified according to their incremental complexity with respect to locally persistent algorithms. The computational-complexity hierarchy for dynamic problems is depicted in Figure 11. In the remainder of this section we describe the results from other papers that have a bearing on this way of classifying dynamic problems; some additional discussion of these problems can be found in Section 6.

The problem of incremental attribute evaluation for noncircular attribute grammars—how to reevaluate the attributes of an attributed derivation trees after a restructuring operation has been applied to the tree—was shown by Reps to be linear in $\|\delta\|$ [31] (see also [32] and [33]). Thus, the algorithm he gave for the problem is asymptotically optimal. ([31] is also the first paper that we are aware of in which an incremental algorithm is analyzed in terms of the parameter $\|\delta\|$.)

The concept of a locally persistent algorithm is due to Alpern *et al.* [1]. Alpern *et al.* also established two results concerning the performance of incremental algorithms in terms of the parameter $\|\delta\|$. Their results concerned two problems: the dynamic circuit-annotation problem and the problem of maintaining a priority ordering in a DAG. In the dynamic priority-ordering problem, as the DAG is modified the goal is to maintain priorities on the graph vertices such that if there is a path from v to w then



MEET−SEMILATTICE DATA−FLOW ANALYSIS PROBLEMS [Section 4.3]
SINGLE−SOURCE/SINK CLOSED−SEMIRING PATH PROBLEMS [Section 4.2]
SINGLE−SOURCE/SINK SHORTEST PATH (>= 0) [Section 4.2]
UNBOUNDED
SINGLE−SOURCE/SINK REACHABILITY [Section 4.1]
EXPONENTIAL
$\|\delta\|\ 2^{\|\delta\|}$
WEIGHTED CIRCUIT−ANNOTATION PROBLEM ([29])
$2^{\|\delta\|}$
CIRCUIT−ANNOTATION PROBLEM [1], [Section 3.3]
POLYNOMIAL
PRIORITY ORDERING [1]
$\|\delta\| \log \|\delta\|$
ALL−PAIRS SHORTEST PATH (> 0) [Section 3.2]
SINGLE−SOURCE/SINK SHORTEST PATH (> 0) [Section 3.1]
$\|\delta\|$
ATTRIBUTE UPDATING [31]

Figure 11. The computational-complexity hierarchy for dynamic problems that exists when problems are classified according to their incremental complexity in terms of the parameter $\|\delta\|$ with respect to locally persistent algorithms. (In the figure, we do not distinguish between $\|\delta\|_1$ and $\|\delta\|_2$. $\|\delta\|$ represents $\|\delta\|_1$ in all cases except for APSP>0.)

priority$(v) >$ *priority*(w).[4] Alpern *et al.* established the following results concerning these problems:

(1) They showed that, with both edge insertions and deletions permitted, the problem of maintaining priorities in a DAG (as well as determining whether an edge insertion introduces any cycles) can be solved in time $O(\|\delta\|^2 \log \|\delta\|)$. Their algorithm processes unit changes in time $O(\|\delta\| \log \|\delta\|)$.

(2) They showed that any locally persistent incremental algorithm for the dynamic circuit-annotation problem is $\Omega(2^{\|\delta\|})$. (Each function associated with a vertex is assumed to be computable in unit time.)

The latter result separates the class of inherently exponentially bounded dynamic problems from the class of polynomially bounded dynamic problems. Recall that in Section 3.3 we give a bounded algorithm for the dynamic circuit-annotation problem (where the bound is an exponential function of $\|\delta\|$). Previous to our work, no bounded algorithm for the dynamic circuit-annotation problem was known.

6. Relation To Previous Work

A key contribution of this paper is that it sheds light on the general problem of analyzing the computational complexity of incremental algorithms. Our work is based on the idea of measuring the cost of an incremental algorithm in terms of the parameter $\|\delta\|$—which is related to the sum of the sizes of the *changes* in the input and the output—rather than in terms of the size of the *entire* (current) input. The advantage of

[4]Note that the dynamic priority-ordering problem is somewhat different from the other problems we have looked at in that the priority-ordering problem concerns a *relation* on graph vertices and labels, rather than a *function* from vertices to labels; that is, many labelings are possible for a given graph. For problems like this, Alpern *et al.* define $\|\delta\|$ to be the size of the minimal change to the current labeling needed to reach any of the solutions for the modified graph.

this approach is that an analysis in terms of $\| \delta \|$ characterizes how well an algorithm performs relative to the amount of work that absolutely must be performed. The paper presents new upper-bound results as well as new lower-bound results. Together with some previously known results, our results help one to understand the complexity hierarchy that exists when dynamic problems are classified according to their incremental complexity with respect to locally persistent algorithms.

Ryder and Paull have remarked about the “inappropriateness of worst-case analysis for dynamic algorithms”[40]; similar remarks have appeared in several other papers. However, our work shows that for some dynamic problems it is not that *worst-case analysis* is inappropriate, but rather that an analysis carried out in terms of the parameter $|input|$ is inappropriate. For example, when the cost of the computation is expressed as a function of $|input|$, in the worst case no incremental algorithm for SSSP>0 can perform better than the best batch algorithm; however, we have shown that there is an incremental algorithm for SSSP>0 with (worst-case) performance $O(\| \delta \| + |\delta| \log |\delta|)$.

The remainder of this section discusses how our results relate to previous work on incremental computation and incremental algorithms.

6.1. Previous Work on Classifying Incremental Problems

The problem of classifying dynamic problems has been addressed in two previous papers, one by Reif [30] and one by Berman, Paull, and Ryder [3]. One aspect of our work that sets it apart from both of these papers is that we analyze incremental complexity in terms of the adaptive parameter $\| \delta \|$, rather than in terms of the size of the current input.

The paper by Reif primarily concerns an algorithm for the connectivity problem in undirected graphs when edge deletions but not edge insertions are permitted [30].[5] At the end of the paper, Reif lists a number of dynamic problems “. . . with linear time sequential RAM algorithms on a single input instance, but which seem to require a complete recomputation in the worst case if a single symbol of the input is modified.” He observes that the problems on his list are not only interreducible, but that the reductions meet two properties:

(P1) The reductions between problems in the group can be performed in linear time by a sequential RAM.

(P2) There exist suitable encodings such that if one symbol of input to an already computed reduction is modified, the reduction can be updated in constant time by a sequential RAM.

Reif draws the following conclusion:

> Consider the dynamic problem of processing a sequence of n single bit modifications to an input instance of . . . size n, where the problem satisfies (P1) and (P2). It follows from (P1) and (P2) that if any of the resulting dynamic problems can be solved in $t(n) = o(n^2)$ time, then all these dynamic problems can be solved in $O(t(n))$ time.

Whereas Reif considers reductions between dynamic *decision* problems, the reductions that we present in Section 4 are among dynamic *optimization* problems (where the output is a set or a mapping). Because our goal is to characterize incremental complexity in terms of the parameter $\| \delta \|$, an additional property is needed beyond that of “linear-time reducibility with constant-time updatability” (*i.e.*, P1 and P2). To see why, suppose that there exists an $O(f(\| \delta_B \|))$ updating algorithm for problem *B*, where *f* is a polynomial. In order to guarantee that a reduction of problem *A* to problem *B* provides an $O(f(\| \delta_A \|))$ updating algorithm for problem *A*, problem encodings and reductions must meet the following property (in

---

[5]The only other permitted operations are queries of the form “Does there exist a path in the current graph between two given vertices?”, which must be answered on-line. For a graph in which the sum of the number of vertices and edges is n on which one performs n operations, Reif’s algorithm has total cost $O(ng + n \log n)$, where g is the genus of the graph.

addition to P1 and P2):

(P3) For every instance I_A of problem A and modification δ_A, the parameter $\| \delta_B \|_{I_B}$ must be $O(\| \delta_A \|_{I_A})$, where I_B is the transformed form of I_A and δ_B is the image of modification δ_A.

As in much of the previous work on incremental computation, Reif is concerned with measuring incremental complexity in terms of the size of the input, whereas in this paper we explore the consequences of measuring incremental complexity in terms of $\| \delta \|$. On the other hand, it is not clear that it would make sense to measure the complexity of dynamic decision problems in terms of $\| \delta \|$.

A different approach to the problem of classifying dynamic problems was proposed in a paper by Berman, Paull, and Ryder [3]. Berman, Paull, and Ryder classify dynamic problems through the notion of an *incremental relative lower bound* (IRLB). An IRLB relates the worst-case time required for an dynamic problem to the running time of the time-optimal algorithm for the batch problem. Some of the differences between their approach and ours are as follows:

(1) Whereas the work of Berman, Paull, and Ryder establishes *relative* lower bounds for dynamic problems, our work concerns “inherent” complexity bounds, that is, bounds expressed in terms of some parameter of the problem itself. (In addition, we discuss upper bounds as well as lower bounds.)

(2) The results of Berman, Paull, and Ryder on IRLB’s are expressed in terms of the time required by the time-optimal algorithm for the corresponding batch problem and, in some cases, the size of the input. Our results are expressed in terms of $\| \delta \|$.

It is only fair to point out that both our work and the work of Berman, Paull, and Ryder fail to address adequately the issue of the use and maintenance of auxiliary information by an incremental algorithm. Such information is crucial to the performance of some incremental algorithms, such as Reps’s algorithm for updating the attributes of an attributed tree after a tree modification [31-33]. A second example is the incremental string-matching algorithm described in Section 3.3 of Berman, Paull, and Ryder’s paper, which falls outside the class of incremental algorithms for which their bounds apply because of the amount of auxiliary information that the algorithm stores and maintains.

In Berman, Paull, and Ryder’s classification scheme, the class of problems with $O(1)$ IRLB’s is the class with the poorest incremental behavior. For these problems, it is possible to show that a single modification, such as the insertion or deletion of a single edge in a graph, can change the problem to one whose solution shares nothing in common with the solution of the original problem (thereby reducing the batch problem to a “one-shot” dynamic problem).[6] Thus, in the worst case, an incremental algorithm for a problem with an $O(1)$ IRLB cannot perform better than the best batch algorithm for the problem. However, this merely leaves us with the following conundrum: “In what sense is a proposed incremental algorithm an improvement over the (best) batch algorithm?”—or more generally, “How does one compare different incremental algorithms for a given problem, if they all have equally bad worst-case behavior (*i.e.*, equally bad when their cost is measured in terms of the size of the current input)?”

Our work shows that if you measure work relative to the amount of work that absolutely must be performed, the picture looks somewhat different. In other words, expressing the cost of an incremental algorithm in terms of the parameter $\| \delta \|$ can sometimes be a fruitful way to compare different algorithms for a problem with an $O(1)$ IRLB (thereby leading to a way out of the conundrum).

---

[6]The arguments that Berman, Paull, and Ryder use to establish relative lower bounds for various problems are similar to the ones used by Spira and Pan [43] and Even and Gazit [15] to establish that no incremental algorithm for the all-pairs shortest-path problem can do better in the worst case than the best batch algorithm for the problem.

Although knowing that a problem has an IRLB of $O(1)$ is certainly a property of interest (since the knowledge that there are modifications for which an incremental algorithm will perform no better than the best batch algorithm answers the question "How bad can things get?"), we believe that our work demonstrates that the notion of an $O(1)$ IRLB does not characterize the class of problems with inherently poor incremental performance. In particular, using an argument of the kind given by Berman, Paull, and Ryder, we can show that SSSP>0 is in the class of problems with $O(1)$ IRLB's:

> Given an input graph $G = (V, E)$ for SSSP>0, modify G by adding a new vertex v to V. For each vertex $v_i \in V - \{ v, sink(G) \}$, add an edge $v_i \longrightarrow v$ with weight $k/3$, where k is the length of the shortest edge in the original graph whose target is $sink(G)$; in addition, add an edge $v \longrightarrow sink(G)$, also with weight $k/3$. This construction can be carried out in $\Theta(|V|)$ steps. The solution of SSSP>0 for the modified graph is immediate: $dist(sink(G)) = 0$, $dist(v) = k/3$, and for each vertex $w \in V - \{ v, sink(G) \}$, $dist(w) = 2k/3$ (since the shortest path from each such vertex w to $sink(G)$ is $[w, v, sink(G)]$). To create a graph that has the same solution as the original graph, we merely have to remove a single edge, namely $v \longrightarrow sink(G)$. Thus, we conclude that SSSP>0 has an IRLB of $O(1)$.

However, as we have shown in Section 3.1 of this paper, there is a bounded incremental algorithm for SSSP>0 with time complexity $O(\| \delta \| + |\delta| \log |\delta|)$.

The fact that SSSP>0 has an $O(1)$ IRLB *and* a polynomially bounded incremental algorithm is what leads us to conclude that the notion of an $O(1)$ IRLB does not characterize the class of problems with inherently poor incremental performance. Thus, it is natural to ask: "What does characterize the problems with inherently poor incremental performance?" Although we do not claim to have given such a characterization, we believe that this paper provides a model for how this question might ultimately be resolved:

(1) As shown by the results presented both in this paper and in others, the computational complexity of dynamic problems can sometimes be measured in a more refined manner by measuring costs in terms of the parameter $\| \delta \|$.

(2) For the class of unbounded problems in our hierarchy of dynamic problems, there exist families of modifications for which the amount of updating that must be performed is not related to $\| \delta \|$ by any fixed function. In this paper we have shown the existence of unbounded problems only in a single (and somewhat impoverished) model of incremental computation, namely the model of locally persistent algorithms. This model of incremental computation is flawed because it excludes from consideration any algorithm that makes use of locally stored pointers. These lower bound results, however, do hold with respect to more powerful models of computation. (See [29].) We believe that the class of unbounded problems provides an example of the kind of characterization of the problems with inherently poor incremental performance that one should look for in other (as yet unspecified) models of incremental computation.

6.2. Previously Known Results Where Incremental Complexity is Measured in Terms of $\| \delta \|$

There have been a few previous papers in which incremental complexity has been measured in terms of the parameter $\| \delta \|$.

Attribute Updating

The first paper that we are aware of in which an incremental algorithm is analyzed in terms of $\| \delta \|$ is a paper by Reps [31] (see also [32] and [33]).[7] The problem discussed in that paper is incremental attribute evaluation for noncircular attribute grammars—how to reevaluate the attributes of an attributed derivation tree after a restructuring operation (such as the replacement of a subtree) has been applied to the tree. The

[7]In these papers, the parameter $\| \delta \|$ is referred to as $|\text{AFFECTED}|$.

algorithm given is linear in $\| \delta \|$ and hence asymptotically optimal. Subsequently, other optimal algorithms were given for a variety of attribute-grammar subclasses, *e.g.*, absolutely noncircular grammars [33] and ordered attribute grammars [36, 45].

All of the algorithms cited above are locally persistent. In the case of the algorithms for noncircular attribute grammars and absolutely noncircular attribute grammars, the cost of an operation that moves the editing cursor in the tree is proportional to the length of the path along which the cursor is moved. (It is necessary to perform a unit-cost update to the auxiliary information used by the attribute updating algorithm at each vertex on the path along which the editing cursor is moved.) For ordered attribute grammars, however, a random-access movement of the editing cursor in the tree is a unit-cost operation.

There are also a variety of other attribute-updating algorithms described in the literature, including one that handles k simultaneous subtree replacements in an n-node tree and runs in amortized time $O((\| \delta \| + k) \cdot \log n)$ [34], and another that permits unit-cost, random-access cursor motion for noncircular attribute grammars and runs in amortized time $O(\| \delta \| \cdot \sqrt{n})$ [35]. These algorithms have “hybrid” complexity measures, in the sense that the running time is a function of the size of the current input as well as $\| \delta \|$ (*i.e.*, the running time is of the form $O(f(|input|, \| \delta \|))$.

Priority Ordering and the Circuit-Value Problem

A paper by Alpern *et al.* [1] concerning the dynamic circuit-value problem and the problem of maintaining a priority ordering in a DAG presents results on the incremental complexity of both problems in terms of the parameter $\| \delta \|$. The results from their work that are related to the ideas presented in this paper are as follows:

(1) They showed that, with both edge insertions and deletions permitted, the problem of maintaining priorities in a DAG (as well as determining whether an edge insertion introduces any cycles) can be solved in time $O(\| \delta \|^2 \log \| \delta \|)$. In the case of unit changes, their algorithm runs in time $O(\| \delta \| \log \| \delta \|)$.

(2) They defined the concept of a locally persistent incremental algorithm, and showed that a lower-bound on any locally persistent algorithm for the dynamic circuit-value problem is $\Omega(2^{\| \delta \|})$.

(3) They gave an (unbounded) algorithm for the dynamic circuit-value problem that used their dynamic priority-ordering algorithm as a subroutine. In this algorithm, after a change to the graph, first priorities are updated; then, vertex re-evaluations are scheduled (via a worklist algorithm that uses a priority queue for the worklist). This algorithm runs in time

$$\| \delta_{PriorityOrdering} \|^2 \log \| \delta_{PriorityOrdering} \| + \| \delta_{CircuitValue} \| \log \| \delta_{CircuitValue} \| .$$

Because the quantity $\| \delta_{PriorityOrdering} \|$ is not bounded by any function of $\| \delta_{CircuitValue} \|$, this algorithm for the dynamic circuit-value problem is unbounded.

6.3. Previous Work on Incremental Shortest-Path Algorithms

Section 3.2 of this paper presents a bounded algorithm for the dynamic all-pairs shortest-path problem with positive edge weights (APSP>0) (assuming the collection of vertices is fixed in advance). Previous to this work no bounded algorithm was known for updating the solution to the all-pairs shortest-path problem after the deletion of an edge. Though they do not use the concept of boundedness, Rohnert [38], Even and Gazit [15], Lin and Chang [21], and Ausiello *et al.* [2] do provide bounded algorithms for updating the solution to the all-pairs shortest-path problem after the insertion of an edge.

There have been three previous papers on handling edge deletion in APSP>0—by Dionne [11], Rohnert [38], and Even and Gazit [15]—in which the analysis might be misinterpreted, on first reading, as demonstrating that the algorithms are bounded. In fact, the algorithms given in all three papers have *unbounded* incremental complexity in general.

As we stated in the introduction, it is important not to confuse $\| \delta \|$, which characterizes the amount of work that it is absolutely necessary to perform for a given dynamic *problem*, with quantities that reflect the updating costs for various internal data structures that store auxiliary information used by a particular *algorithm* for the dynamic problem. (Although costs of the latter sort do, in some sense, reflect "the size of the change," they do not represent an updating cost that is inherent to the dynamic problem itself; one must ask how these costs compare with $\| \delta \|$.) For example, with both Rohnert's and Even and Gazit's algorithms for edge-deletion, the total updating cost depends on potentially unbounded costs that arise because of the need to update various data structures used in the two algorithms. By contrast, in our algorithm for APSP>0 all costs are bounded by $\| \delta \|$, *including all costs for updating the data structures used by the algorithm*.

In addition to maintaining the distance matrix for the graph, many of the incremental algorithms for the all-pairs shortest-path problem are also capable of handling requests of the form "List a shortest path from vertex *x* to vertex *y*" in time proportional to the number of vertices in the path reported by the algorithm. Our procedures $\text{DeleteEdge}_{\text{APSP>0}}$ and $\text{InsertEdge}_{\text{APSP>0}}$ can be generalized to maintain one shortest path between any pair of vertices without increasing their asymptotic time complexity.

Other previous work on how to maintain shortest paths in graphs incrementally includes papers by Murchland [25, 26], Loubal [22], Rodionov [37], Halder [18], Pape [27], Hsieh *et al.* [20], Cheston [8], Goto *et al.* [16], Cheston and Corneil [9]; however, none of the papers in this group analyze either the single-source or the all-pairs problem in terms of the parameter $\| \delta \|$, and furthermore, when such an analysis is made, none of the algorithms presented in these papers turns out to be bounded.

6.4. Other Related Work

For batch algorithms, the concept of measuring the complexity of an algorithm in terms of the sum of the sizes of the input and the output has been explored by Cai and Paige [4, 5] and by Gurevitch and Shelah [17]. In this paper, we measure the complexity of an *dynamic* algorithm in terms of the sum of the sizes of the *changes* in the input and the output.

A number of papers in the literature on dynamic algorithms concern incremental data-flow analysis [6, 23, 24, 39, 40, 46]. However, only one other paper has ever examined the question of whether incremental data-flow analysis is, in any sense, an "intrinsically hard" problem: Berman, Paull, and Ryder show that a number of incremental data-flow analysis problems have $O(1)$ IRLB's [3], which puts them in the class of problems with the poorest incremental behavior (in the sense of Berman, Paull, and Ryder). On the other hand, what an $O(1)$ IRLB signifies is merely that the worst-case behavior of a dynamic algorithm for such a problem can be no better than that of the best algorithm for the batch version of the problem. In addition, the fact that SSSP>0 has an $O(1)$ IRLB *and* a bounded dynamic algorithm re-opens the question of whether incremental data-flow analysis really *is* inherently difficult. Our results from Section 4 show that, under the model of locally persistent algorithms, incremental data-flow analysis problems are unbounded—and hence in this model they *are* inherently difficult problems. (However, this model is a very restricted model of incremental computation, and the question is open as to whether there exist any bounded incremental data-flow analysis algorithms outside the class of locally persistent algorithms.)

To establish lower bounds on dynamic problems, it is necessary to have a model of incremental computation. In this paper all lower-bound results apply to the locally persistent algorithms, a model of incremental computation that was defined by Alpern et al. [1]. The paper by Spira and Pan that establishes lower bounds on updating minimum spanning trees and single-source shortest paths in positively weighted graphs makes an assumption that is not exactly the same as restricting attention to only locally persistent algorithms, but is similar in spirit:

. . . we have discussed updating where only the answer to the problem considered is retained. It seems likely that if intermediate information in obtaining the original solution is kept, improvements will be possible. We have not investigated this. ([43], p. 380).

What is unsatisfactory about these models of incremental computation is that, at best, only very limited use of auxiliary storage is permitted. Berman, Paull, and Ryder do discuss a model of incremental computation that has somewhat fewer restrictions on the use of auxiliary storage [3]; however, in their model the cost of initializing any auxiliary storage used must be less than the cost of running the optimal-time batch algorithm for the problem. There are certainly reasonable dynamic algorithms that, because of the amount of auxiliary information that the algorithms store and maintain, lie outside the class of algorithms covered by Berman, Paull, and Ryder's model. (For instance, see Section 3.3 of [3].) Thus, a desirable goal for future research is to develop a better model of incremental computation that better addresses the issue of the use and maintenance of auxiliary storage by dynamic algorithms.

As a final closing remark, it should be noted that although most dynamic algorithms that have been proposed are unbounded in the sense of the term used in this paper, from a practical standpoint such algorithms may give satisfactory performance in real systems. For instance, Hoover presents evidence that his unbounded algorithm for the circuit-value problem performs well in practice [19]; Ryder, Landi, and Pande present evidence that the unbounded incremental data-flow analysis algorithm of Carroll and Ryder [6] performs well in practice [41]; Dionne reports excellent performance for some unbounded algorithms for APSP>0 [11].

REFERENCES

1. Alpern, B., Hoover, R., Rosen, B.K., Sweeney, P.F., and Zadeck, F.K., "Incremental evaluation of computational circuits," pp. 32-42 in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms,* (San Francisco, CA, Jan. 22-24, 1990), Society for Industrial and Applied Mathematics, Philadelphia, PA (1990).
2. Ausiello, G., Italiano, G.F., Spaccamela, A.M., and Nanni, U., "Incremental algorithms for minimal length paths," pp. 12-21 in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms,* (San Francisco, CA, Jan. 22-24, 1990), Society for Industrial and Applied Mathematics, Philadelphia, PA (1990).
3. Berman, A.M., Paull, M.C., and Ryder, B.G., "Proving relative lower bounds for incremental algorithms," *Acta Informatica* **27** pp. 665-683 (1990).
4. Cai, J. and Paige, R., "Binding performance at language design time," pp. 85-97 in *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages,* (Munich, W. Germany, January 1987), ACM, New York, NY (1987).
5. Cai, J. and Paige, R., "Languages polynomial in the input plus output," in *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology (AMAST),* (Iowa City, Iowa, May 22-25, 1991), (1991).
6. Carroll, M. and Ryder, B., "Incremental data flow update via attribute and dominator updates," pp. 274-284 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).
7. Carroll, M.D., "Data flow update via dominator and attribute updates," Ph.D. dissertation, Rutgers University, New Brunswick, NJ (May 1988).
8. Cheston, G.A., "Incremental algorithms in graph theory," Ph.D. dissertation and Tech. Rep. 91, Dept. of Computer Science, University of Toronto, Toronto, Canada (March 1976).
9. Cheston, G.A. and Corneil, D.G., "Graph property update algorithms and their application to distance matrices," *INFOR* **20**(3) pp. 178-201 (August 1982).
10. Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms,* MIT Press, Cambridge, MA (1990).
11. Dionne, R., "Etude et extension d'un algorithme de Murchland," *INFOR* **16**(2) pp. 132-146 (June 1978).
12. Driscoll, J.R., Gabow, H.N., Shrairman, R., and Tarjan, R.E., "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Communications of the ACM* **31**(11) pp. 1343-1354 (1988).

13. Edmonds, J. and Karp, R.M., "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM* **19** pp. 248-264 (1972). As cited in reference [45].

14. Even, S. and Shiloach, Y., "An on-line edge-deletion problem," *J. ACM* **28**(1) pp. 1-4 (January 1981).

15. Even, S. and Gazit, H., "Updating distances in dynamic graphs," pp. 271-388 in *IX Symposium on Operations Research,* (Osnabrueck, W. Ger., Aug. 27-29, 1984), *Methods of Operations Research*, Vol. 49, ed. P. Brucker and R. Pauly,Verlag Anton Hain (1985).

16. Goto, S. and Sangiovanni-Vincentelli, A., "A new shortest path updating algorithm," *Networks* **8**(4) pp. 341-372 (1978).

17. Gurevich, Y. and Shelah, S., "Time polynomial in input or output," *J. Symbolic Logic* **54**(3) pp. 1083-1088 (September 1989).

18. Halder, A.K., "The method of competing links," *Transportation Science* **4** pp. 36-51 (1970).

19. Hoover, R., "Incremental graph evaluation," Ph.D. dissertation and Tech. Rep. 87-836, Dept. of Computer Science, Cornell University, Ithaca, NY (May 1987).

20. Hsieh, W., Kershenbaum, A., and Golden, B., "Constrained routing in large sparse networks," pp. 38.14-38.18 in *Proceedings of IEEE International Conference on Communications*, , Philadelphia, PA (1976).

21. Lin, C.-C. and Chang, R.-C., "On the dynamic shortest path problem," *Journal of Information Processing* **13**(4)(1990).

22. Loubal, P., "A network evaluation procedure," *Highway Research Record* **205** pp. 96-109 (1967).

23. Marlowe, T.J., "Data flow analysis and incremental iteration," Ph.D. dissertation and Tech. Rep. DCS-TR-255, Rutgers University, New Brunswick, NJ (October 1989).

24. Marlowe, T.J. and Ryder, B.G., "An efficient hybrid algorithm for incremental data flow analysis," pp. 184-196 in *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages,* (San Francisco, CA, Jan. 17-19, 1990), ACM, New York, NY (1990).

25. Murchland, J.D., "The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph," Tech. Rep. LBS-TNT-26, London Business School, Transport Network Theory Unit, London, UK (1967).

26. Murchland, J.D., "A fixed matrix method for all shortest distances in a directed graph and for the inverse problem," Doctoral dissertation, Universität Karlsruhe, Karlsruhe, W. Germany ().

27. Pape, U., "Netzwerk-veraenderungen und korrektur kuerzester weglaengen von einer wurzelmenge zu allen anderen knoten," *Computing* **12** pp. 357-362 (1974).

28. Ramalingam, G. and Reps, T., "On the computational complexity of incremental algorithms," TR-1033, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1991).

29. Ramalingam, G., "Bounded Incremental Computation," Ph.D. dissertation and Tech. Rep. TR-1172, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1993).

30. Reif, J.H., "A topological approach to dynamic graph connectivity," *Information Processing Letters* **25**(1) pp. 65-70 (1987).

31. Reps, T., "Optimal-time incremental semantic analysis for syntax-directed editors," pp. 169-176 in *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM, January 25-27, 1982), ACM, New York, NY (1982).

32. Reps, T., Teitelbaum, T., and Demers, A., "Incremental context-dependent analysis for language-based editors," *ACM Trans. Program. Lang. Syst.* **5**(3) pp. 449-477 (July 1983).

33. Reps, T., *Generating Language-Based Environments,* The M.I.T. Press, Cambridge, MA (1984).

34. Reps, T., Marceau, C., and Teitelbaum, T., "Remote attribute updating for language-based editors," pp. 1-13 in *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages*, (St. Petersburg, FL, Jan. 13-15, 1986), ACM, New York, NY (1986).

35. Reps, T., "Incremental evaluation for attribute grammars with unrestricted movement between tree modifications," *Acta Informatica,* pp. 155-178 (1988).

36. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors,* Springer-Verlag, New York, NY (1988).

37. Rodionov, V., "The parametric problem of shortest distances," *U.S.S.R. Computational Math. and Math. Phys.* **8**(5) pp. 336-343 (1968).

38. Rohnert, H., "A dynamization of the all pairs least cost path problem," pp. 279-286 in *Proceedings of STACS 85: Second Annual Symposium on Theoretical Aspects of Computer Science,* (Saarbruecken, W. Ger., Jan. 3-5, 1985), *Lecture Notes in Computer Science*, Vol. 182, ed. K. Mehlhorn,Springer-Verlag, New York, NY (1985).

39. Rosen, B.K., “Linear cost is sometimes quadratic,” pp. 117-124 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), ACM, New York, NY (1981).

40. Ryder, B.G. and Paull, M.C., “Incremental data flow analysis algorithms,” *ACM Trans. Program. Lang. Syst.* **10**(1) pp. 1-50 (January 1988).

41. Ryder, B.G., Landi, W., and Pande, H.D., “Profiling an incremental data flow analysis algorithm,” *IEEE Transactions on Software Engineering* **SE-16**(2)(February 1990).

42. Sedgewick, R., *Algorithms,* Addison-Wesley, Reading, MA (1983).

43. Spira, P.M. and Pan, A., “On finding and updating spanning trees and shortest paths,” *SIAM J. Computing* **4**(3) pp. 375-380 (September 1975).

44. Tarjan, R.E., *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, PA (1983).

45. Yeh, D., “On incremental evaluation of ordered attributed grammars,” *BIT* **23** pp. 308-320 (1983).

46. Zadeck, F.K., “Incremental data flow analysis in a structured program editor,” *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction,* (Montreal, Can., June 20-22, 1984), *ACM SIGPLAN Notices* **19**(6) pp. 132-143 (June 1984).