ON THE COMPUTATIONAL COMPLEXITY OF INCREMENTAL ALGORITHMS

by

G. Ramalingam and Thomas Reps

Computer Sciences Technical Report #1033

August 1991

On the Computational Complexity of Incremental Algorithms

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 often not very informative. (By an "incremental algorithm," we mean an algorithm that makes use of the solution to one problem instance to find the solution to a "nearby" problem instance.) When the cost of the computation is expressed as a function of the size of the (current) input, many 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 incremental problems.

The paper presents new upper-bound results as well as new lower-bound results. First, three problems—the single-sink shortest-path problem with positive edge weights, the all-pairs shortest-path problem with positive edge weights, and the circuit-value problem—are shown to have bounded incremental complexity (*i.e.*, incremental complexity bounded by a function of the sum of the sizes of the changes in the input and the output). The single-sink shortest-path problem with positive edge weights and the all-pairs shortest-path problem with positive edge weights are shown to be P-time incremental; the circuit-value problem is shown to be *Exp*-time incremental. We then turn to lower-bounds and establish a number of lower bounds with respect to a class of algorithms called the locally persistent algorithms. We first demonstrate the existence of a non-incremental problem (*i.e.*, a problem for which *no* bounded locally persistent incremental algorithm exists). We then demonstrate that a number of other problems, including the closed-semiring path problems in directed graphs and the meet-semilattice data-flow analysis problems, are non-incremental with respect to the class of locally persistent algorithms.

Our results, together with some previously known ones, 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. In particular, these results separate the classes of P-time incremental problems, inherently *Exp*-time incremental problems, and non-incremental problems.

Categories and Subject Descriptors: E.1 [**Data Structures**] -- *graphs*; F.1.3 [**Computation by Abstract Devices**]: Complexity Classes -- *complexity hierarchies, relations among complexity classes*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems -- *computations on discrete structures*; G.2.1 [**Discrete Mathematics**]: Combinatorics -- *combinatorial algorithms*; G.2.2 [**Discrete Mathematics**]: Graph Theory -- *graph algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: all-pairs shortest-path problem, circuit-value problem, closed-semiring path problems, incremental algorithm, locally persistent algorithm, meet-semilattice data-flow analysis problems, reachability problem, single-sink shortest-path problem

---

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' address: Computer Sciences Department, University of Wisconsin–Madison, 1210 W. Dayton St., Madison, WI 53706.
E-mail: {ramali, reps}@cs.wisc.edu

1. INTRODUCTION

Many kinds of interactive systems developed in the last fifteen years, such as word processors, electronic spreadsheets, language-based program editors, and systems for computer-aided design, are examples of what has been called the *continuous-execution* [18] or *immediate-computation* [34] paradigm. In a system that follows the immediate-computation paradigm, the application program and the editing function are closely coupled. Each modification to a datum is processed by the "application" and has essentially instantaneous effect. An important consequence is that the computation that is the "essence" of the application program is always consistent with the current state of the data, thereby providing useful immediate feedback to the user as the data are manipulated.

Extra computation is generally required in a software system that follows the immediate-computation paradigm because the data pass through many intermediate states that would not arise in a batch-mode computation. These extra steps are often acceptable, however, because an immediate-computation system can exploit the surplus processing capacity of single-user workstations. (Whereas the unused cycles of time-shared computers are at the disposal of others, the spare cycles of single-user computers go to waste.)

A fundamental problem that confronts the builder of an interactive system based on the immediate-computation paradigm is the problem of *incremental change*—the system must be designed to provide its services efficiently as the user makes a sequence of small changes. Users' needs for rapid response in the face of small changes makes the study of *incremental computation* and *incremental algorithms* very important.

Although by now there is a considerable literature on incremental computation and incremental algorithms, with a few exceptions these papers have addressed only very specific incremental-computation problems. The amount of progress in formulating general principles toward a theory of incremental computation has been disappointing. A key contribution of this paper is that it *does* shed light on a general problem, namely, the analysis of the computational complexity of incremental-computation problems.

The utility of the approach that we develop in this paper to the general problem of complexity analysis for incremental problems is illustrated by a number of new results on several specific incremental-computation problems. In addition to their role in illustrating the existence of a computational-complexity hierarchy for incremental problems (see Figure 10), these results are of considerable interest in their own right because the problems studied are at the core of some current—as well as potential future—interactive application programs (*e.g.*, spreadsheets, CAD tools, and program-analysis tools). For instance, our work shows that incremental data-flow analysis is, in some sense, an *intrinsically hard* problem (see Sections 4.3 and 6).

The abstract problem of incremental computation in immediate-computation systems can be phrased informally as follows: The application program computes a function f on the user's "input" data x, where x is some data structure, such as a tree, graph, or matrix, representing the "document" that the user is preparing. 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 to some space of values (for example, a mapping from spreadsheet cells to values). When the user changes the input data from x to x', the application program must compute a new annotation $f(x')$ in place of $f(x)$. Of course, $f(x')$ could be calculated from scratch, but the danger is that this will be too slow to provide adequate response time in an interactive system. However, because the increment from x to x' is often small—for example, when x and x' represent "nearby" problem instances related by a small modification made to x by the user—the increment from $f(x)$ to $f(x')$ is often small as well. Thus, it is desirable to employ an algorithm that re-uses old information so as to avoid as much recomputation as possible. An algorithm that uses information from the old computation of $f(x)$ to compute the new annotation $f(x')$ is called an *incremental algorithm.*

For batch-mode computation, 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 incremental computation, this kind of analysis is often not very informative; when the cost of the computation is expressed as a function of the size of the (current) input (*i.e.*, $|x'|$), many 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 [7, 8, 19, 24, 48]. 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 an incremental-computation *problem* (as opposed to an analysis of a particular incremental *algorithm* for a problem), demonstrating that in the worst-case *no* incremental algorithm for the problem can perform asymptotically better than the time required to perform the computation from scratch [3, 14, 43]. For these reasons, worst-case analysis (with costs expressed as a function of the size of the input) has not been of much help for making comparisons between different incremental algorithms.

However, as discussed in this paper, there is an alternative approach that can be taken to analyze the behavior of incremental algorithms. Because an incremental algorithm makes use of the solution to one problem instance to find the solution to a "nearby" problem instance, a useful parameter for evaluating the time complexity of an incremental algorithm is the sum of the sizes of the *changes* in the input and output. It is this approach to analyzing the computational complexity of incremental algorithms that is the subject of this paper: instead of analyzing the behavior of algorithms in terms of the size of the *entire* current input (*i.e.*, $|x'|$), we concentrate on analyzing algorithms in terms of the size of an "adaptive" parameter, denoted by CHANGED, that captures the changes in the input and output. Thus, analysis in terms of CHANGED characterizes the behavior of an algorithm in terms of the amount of work that it is absolutely necessary to perform for the given incremental problem.

For the moment, we define CHANGED informally as CHANGED = $\Delta\,input + \Delta\,output$, where $\Delta\,input$ captures the differences between data structures x and x' and $\Delta\,output$ captures the differences between solutions $f(x)$ and $f(x')$.[1] The size of CHANGED following an input change δ will be denoted by $\|\delta\|$. Our goal is to analyze the time complexity of incremental algorithms in terms of $\|\delta\|$. This change in approach allows us to develop a more informative theory of computational complexity for incremental problems.

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 incremental *problem*, with quantities that reflect the updating costs for various internal data structures that store auxiliary information used by a particular *algorithm* for the incremental problem. The parameter CHANGED represents the updating costs that are *inherent to the incremental problem* itself. CHANGED is an *adaptive parameter*: in the worst case—when the entire output is affected—CHANGED equals $\Delta\,input + output$; in the best case—when there is no change in the output—CHANGED equals $\Delta\,input$.

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

---

[1] We will be more precise about the definition of CHANGED in Section 2; note that CHANGED must be a function of both the current object and the change applied to the object.

It is appropriate at this point to mention the notion of amortized-cost analysis of algorithms. Here is Tarjan on the subject of amortization [45]:

> Amortization is appropriate in situations where particular algorithms are repeatedly applied, as occurs with operations on data structures. By averaging the time per operation over a worst-case sequence of operations, we sometimes can obtain an overall time bound much smaller than the worst-case time per operation multiplied by the number of operations [45].

There are incremental problems for which the benefits mentioned above are realized. Examples include the results of Even and Shiloach [13], Reif [29] and Ausiello *et al.* [2]. However, the question of amortized-cost analysis versus worst-case analysis is really orthogonal to the question studied in this paper. What this paper shows is that it can be fruitful to analyze the computational 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 this paper (for an exception, see the footnote near the beginning of Section 3.2), in principle there could exist problems for which a better bound (in terms of $\|\delta\|$) would be obtained if amortized analysis were used.

Although in some worst-case situations, an incremental algorithm can perform asymptotically no better than starting over from scratch,[2] an analysis in terms of $\|\delta\|$ provides information about the adaptive behavior of an algorithm by characterizing how well an algorithm performs relative to the amount of work that it is absolutely necessary to perform. For example, an incremental algorithm that is linear in $\|\delta\|$ can be said to be *optimal* [30,31]: the cost $|\Delta input|$ can be charged to the user's action that modifies the input from x to x'; the cost of updating the solution from $f(x)$ to $f(x')$ can be no less than $|\Delta output|$. Thus, using an analysis in terms of $\|\delta\|$, an incremental algorithm can be said to be optimal even when there exist worst-case examples in which it is essentially necessary to re-solve the problem from scratch. Under the usual way of analyzing algorithms (*i.e.*, in terms of $|x'|$), the complexity would not even appear to be any better than that of the batch start-over algorithm.

An incremental algorithm is said to be *bounded* if, for all input objects and for all changes that can be applied to an input object, 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* input (*i.e.*, $|x'|$). Otherwise, an incremental algorithm is said to be *unbounded*. A problem is said to be *incremental* (non-incremental) if it has (does not have) a bounded incremental algorithm.

The notion of a bounded incremental algorithm is not new to this paper. Examples of bounded incremental algorithms that have been presented in previous work include (1) Reps's algorithm for updating the attributes of an attributed tree after an editing modification [30,31], and (2) the algorithm of Alpern *et al.* for maintaining a priority ordering in a DAG [1].

What is new in this paper is that we systematically classify a wide range of problems according to their degree of boundedness. Our results can be summarized as follows:

(1) We establish three new upper-bound results: 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

---

[2]It is true that in such worst-case situations, it would often have been better to have actually started over from scratch and employed the best batch-mode algorithm. There are two reasons for this: (1) Often the best batch-mode algorithm will have superior asymptotic performance to that obtained when the incremental algorithm is applied to the batch problem (*i.e.*, when the incremental algorithm is used in a "start-over" mode in which the algorithm is not supplied any previous solution to a "nearby" problem instance); (2) Even if it does not have superior asymptotic performance, the batch algorithm will usually have a smaller constant factor because there are often overhead costs involved in maintaining auxiliary data structures that are required by the incremental algorithm.

circuit-value problem[3] are shown to have bounded incremental complexity. SSSP>0 and APSP>0 are shown to be *P*-time incremental; the circuit-value problem is shown to be *Exp*-time incremental.

(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 exists *no* bounded locally persistent incremental algorithm (*i.e.*, that there exist *non-incremental* problems).

The intrinsic interest of these results is heightened by the fact that the class of non-incremental problems contains many natural problems, including ones 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 incremental-computation problems are classified according to their incremental complexity with respect to locally persistent algorithms. In particular, these results separate the classes of *P*-time incremental problems, inherently *Exp*-time incremental problems, and non-incremental problems. The computational-complexity hierarchy for incremental problems is depicted in Figure 10 (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 *P*-time incremental, yet the very similar problem SSSP≥0 (in which *0-weight* edges are also permitted) is non-incremental.

To appreciate the advantages of analyzing incremental algorithms in terms of the parameter $\|\delta\|$ rather than in terms of the size of the current input, it is instructive to compare our results with those of Spira and Pan [43] concerning the problem SSSP>0. Under an assumption that is somewhat similar to our assumption of locally persistent algorithms, Spira and Pan obtained a lower-bound: they showed that, in the worst-case, *no* incremental algorithm of a certain class can perform better asymptotically than the time required to perform the computation from scratch (where costs are expressed as a function of the size of the current input) [43]. By contrast, we give an algorithm that, in the worst case, is polynomial in $\|\delta\|$ (as opposed to the size of the current input). Because $\|\delta\|$ is a parameter that captures the adaptive behavior of an incremental algorithm, the analysis in terms of $\|\delta\|$ characterizes how well our algorithm performs relative to the amount of work that it is absolutely necessary to perform.

We do not want to give the impression that the notion of boundedness is the only relevant criterion in the study of incremental computation. In fact, relatively few bounded incremental algorithms are actually known, and a number of other vantage points have proved useful in the study of incremental computation. An overview of some of the other approaches that have been taken, together with a discussion of how they relate to the work reported in this paper, can be found in Section 6.

While these other approaches have all contributed to our current understanding of incremental computation, the development of a theory of computational complexity for incremental problems—such as the one presented in this paper—seems to us to be fundamental to gaining a deeper understanding of incremental computation. The problem of classifying incremental problems has also been addressed by Reif [29] and Berman, Paull, and Ryder [3]. In both cases, an important aspect that sets our work apart from theirs is that we analyze incremental complexity in terms of the adaptive parameter $\|\delta\|$, rather than in terms of the size

---

[3] A *circuit* is a DAG with values computed at each vertex. The value computed at a vertex u is a function F_u of the values computed by the predecessors of vertex u. The circuit-value problem is to compute the output value associated with each vertex.

of the current input. A more extensive comparison of our work with Reif's work and Berman, Paull, and Ryder's work can be found in Section 6.1.

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-value 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

In this paper, we concentrate on the class of graph problems that require the computation of some value $S_G(u)$ for each vertex u of the input graph G. $S_G(u)$ is called the (output) value associated with vertex u. For instance, in SSSP>0, $S_G(u)$ is the length of the shortest path from vertex u to a distinguished vertex, denoted by $sink(G)$. In general, some information may be associated with each vertex or edge, such as a real-valued *length* with each edge.

In such problems, the value at a vertex u can typically be computed from the values at the predecessors (or successors) of u. Thus, every vertex u has an associated equation relating the value at u to the values of predecessors (or successors) of u. In general, because the input graphs may be cyclic, the solution desired is some particular fixed point of the collection of equations for the vertices of the graph. (If the input graphs are DAGs, as for the circuit-value problem discussed in Section 3.3, the equations will have a unique solution.)

In the remainder of this section, we briefly describe the graph-theoretic terminology we use and define $\| \delta \|$, the "size of the change in the input and output."

Definition 2.1. A *directed graph* $G = (V(G), E(G))$ consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$. Each edge $b \longrightarrow c \in E(G)$, where $b, c \in V(G)$, is directed from b to 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.

Definition 2.2. Given a directed graph G, a *path* from vertex b to vertex c is a sequence of vertices, $[v_1, v_2, ..., v_k]$, such that: $b = v_1$, $c = v_k$, and $\{ v_i \longrightarrow v_{i+1} \mid i = 1, ..., k-1 \} \subseteq E(G)$. The *length* of the path is $k-1$. (The notion of a path in an undirected graph is defined similarly.)

The sum of the number of vertices and the number of edges in a graph G will be denoted by $|G|$. If K is a set of vertices in a graph G, the cardinality of K will also be denoted by $|K|$. For our purposes, a more useful measure of the "size" of K is the *extended size* of K, which is defined as follows:

Definition 2.3. Let K be a set of vertices in directed graph G. Let $Succ(K)$ denote the set of all vertices that are successors of some vertex in K and $Pred(K)$ denote the set of all vertices that are predecessors of some vertex in K. Let $N(K)$, the neighborhood of K, be the set of all vertices that are in K or are adjacent to some vertex in K: thus, $N(K) = K \cup Succ(K) \cup Pred(K)$. $N^i(K)$ is defined inductively to be $N(N^{i-1}(K))$, where $N^1(K) = N(K)$. The subgraph induced by a set of vertices F will be denoted by $<F>$. The *extended size of K of order i*, denoted by $\| K \|_{i,G}$ is defined to be $|<N^i(K)>|$.

In this paper, we are only ever concerned with extended sizes of order 1 and order 2. $\| K \|_{1,G}$ is the sum of the number of vertices and the number of edges in $<N(K)>$. (Note that $N(K)$ consists of vertices that are at most one "step" away from members of K.) Similarly, $\| K \|_{2,G}$ encompasses the subgraph consist-

ing of all of the vertices and edges that are up to two (undirected) steps away from K. We abbreviate $\| K \|_{1,G}$ as $\| K \|_G$. We also drop the subscript G where it is possible to do so without causing ambiguity.

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 of $G+\delta$ is said to have been *modified* by δ if δ inserted u or some edge incident on u, or deleted some edge incident on u, or modified information associated with u or some edge incident on u. The set of all modified vertices in $G+\delta$ will be denoted by $\text{MODIFIED}_{G,\delta}$. We use $\| \text{MODIFIED}_{G,\delta} \|_{G+\delta}$ as a measure of the size of the change in 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 $\text{AFFECTED}_{G,\delta}$ denote the set of all affected vertices in $G+\delta$. The subscripts of the various sets defined above will be dropped if no confusion is likely. $\| \text{AFFECTED} \|_{i,G+\delta}$ is a measure of the size of the change in output.

We define $\text{CHANGED}_{G,\delta}$ to be $\text{MODIFIED}_{G,\delta} \cup \text{AFFECTED}_{G,\delta}$. $\| \text{CHANGED} \|_{i,G+\delta}$, which we abbreviate to $\| \delta \|_{i,G}$ is a measure of the size of the change in the input and output. (Either or both of the subscripts i and G will be dropped if no confusion is likely; an omitted subscript i implies a value of 1.)

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 $|G|$).[4] 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 incremental (non-incremental) if it has (does not have) a bounded incremental algorithm.

Aside. An alternative strategy for studying the computational complexity of incremental algorithms would have been to restrict the input instances to graphs with a fixed bound on indegree and outdegree, and to express incremental complexity as a function of the parameter $|\delta|_G = |\text{CHANGED}_{G,\delta}|$. Instead, we have chosen to work with problems on general graphs and express incremental complexity in terms of $\| \delta \|_{i,G} = \| \text{CHANGED} \|_{i,G+\delta}$. If one does restrict attention to families of bounded-valence graphs, all complexity bounds given in the paper of the form $O(f(\| \delta \|_{i,G}))$ can be restated as bounds of the form $O(f(|\delta|_G))$, where the constant of proportionality depends on the maximum valence of the graph.
End Aside.

3. UPPER-BOUND RESULTS: THREE BOUNDED INCREMENTAL 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-value problem. SSSP>0 and APSP>0 are shown to be P-time incremental; the circuit-value problem is shown to be *Exp*-time incremental.

---

[4]Note that we use the uniform-cost measure in analyzing the complexity of the steps of an algorithm. Thus, for instance, accessing the successor of a vertex is counted as a unit-cost operation, rather than one with cost $\Omega(\log |V(G)|)$.

3.1. The Incremental Single-Sink Shortest-Path Problem

SSSP>0 can be phrased in terms of two maps: *length*: *edge* $\rightarrow \mathcal{R}^{>0}$ and *dist*: *vertex* $\rightarrow (\mathcal{R}^{\geq 0} \cup \{\infty\})$, where $\mathcal{R}^{>0}$ denotes the set of positive reals and $\mathcal{R}^{\geq 0}$ denotes the set of non-negative reals. Given map *length*, which associates each edge e of G with a positive real number *length*(e), map *dist* is a solution to SSSP>0 iff for every vertex v of G *dist*(v) is the length of the shortest path in G from vertex v to a distinguished vertex *sink*(G), and is ∞ if there is *no* path in G from v to *sink*(G). (It is convenient to think of the *length* and *dist* maps as two real-valued labels associated with each edge and vertex of G, respectively.)

Given graph G, map *length*, solution *dist*, and possibly some auxiliary information, an incremental algorithm for SSSP>0 must update the *dist* map and the auxiliary information after a single edge is deleted from or inserted into the edge set of G. In the case of an edge insertion, the length of the edge is an additional parameter to the insertion procedure. An insertion or deletion of an isolated vertex is also permitted; the corresponding update operation is trivial.

A key aspect of our incremental algorithm for SSSP>0 is that it maintains a distinguished subset of the edges in G's edge set, denoted by $SP(G)$; $SP(G)$ contains exactly those edges that occur in some shortest path from some vertex of G to *sink*(G). That is, the graph $(V(G), SP(G))$ is the graph of shortest paths to *sink*(G). Given graph G and a modification to G, two actions must be carried out: (1) $SP(G)$ must be updated to reflect the edges of G's new shortest-path graph, and (2) the *dist* map must be updated to reflect any changes in the lengths of shortest paths.

Our incremental algorithm for SSSP>0 consists of two procedures, called DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$, which are presented in Figures 1 and 2, respectively.[5]

3.1.1. Deletion of an Edge

The update algorithm for edge deletion is given as procedure DeleteEdge$_{SSSP>0}$ in Figure 1. 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 $v \longrightarrow w$. Note that it is not necessary to update any distances at all unless two conditions hold:

(1) Edge $v \longrightarrow w$ is a member of $SP(G)$.
(2) Edge $v \longrightarrow w$ is the only member of $SP(G)$ whose source is v.

(See lines [1] and [3] of Figure 1.)

Because the *SP* edges form a *DAG*, it is possible to identify the vertices of AFFECTED (*i.e.*, the set of vertices whose shortest distance to *sink*(G) has increased) through a procedure very much like topological sorting: starting from vertex v, a traversal backwards along *SP* edges is carried out, during which each edge traversed is removed from *SP*; a workset is used to record the vertices whose incoming edges are ready to be traversed; a vertex is placed on the workset exactly once, when its count of outgoing *SP* edges drops to 0.

There are two phases involved in updating G (and $SP(G)$) after an edge is deleted. Phase 1 (lines [4]–[15]) carries out the traversal described above and sets the variable AffectedVertices equal to AFFECTED. Phase 2 (lines [16]–[33]) is an adaptation of Dijkstra's (batch) shortest-path algorithm (see reference [10]) that is able to "pick up the process of assigning *dist* values in the middle" by initializing a priority queue appropriately (lines [17]–[21]). It may be easiest to think of Phase 2 as the application of

[5]It is trivial to generalize DeleteEdge$_{SSSP>0}$ to handle the lengthening of an edge and InsertEdge$_{SSSP>0}$ to handle the shortening of an edge.

```
procedure DeleteEdge_{SSSP>0}(G, v → w)
declare
    G: a directed graph
    v → w: an edge to be deleted from G
    WorkSet, AffectedVertices: sets of vertices
    a, b, c, u, v, w, x, y: vertices
    PriorityQueue: a priority queue of vertices
preconditions
    v → w ∈ E(G)
    SP(G) consists of all edges in E(G) on shortest paths to sink(G)
    ∀ v ∈ V(G), dist(v) is length of the shortest path from v to sink(G)
begin
[1]  if v → w ∈ SP(G) then
[2]    Remove edge v → w from SP(G) and from E(G)
[3]    if there does not exist a vertex x such that v → x ∈ SP(G) then
[4]      /* Phase 1: Identify vertices in AFFECTED (the set of vertices whose shortest distance to sink(G) has increased). */
[5]      /*          Set AffectedVertices equal to AFFECTED. */
[6]      WorkSet := { v }
[7]      AffectedVertices := ∅
[8]      while WorkSet ≠ ∅ do
[9]        Select and remove a vertex u from WorkSet
[10]       Insert vertex u into AffectedVertices
[11]       for each vertex x such that x → u ∈ SP(G) do
[12]         Remove edge x → u from SP(G)
[13]         if there does not exist a vertex y such that x → y ∈ SP(G) then Insert vertex x into WorkSet fi
[14]       od
[15]     od
[16]     /* Phase 2: Determine new distances to sink(G) for all vertices in AffectedVertices and update SP(G). */
[17]     PriorityQueue := ∅
[18]     for each vertex a ∈ AffectedVertices do
[19]       dist(a) := min ({ length(a → b) + dist(b) | a → b ∈ E(G) and b ∈ (V(G) − AffectedVertices) } ∪ { ∞ })
[20]       if dist(a) ≠ ∞ then Insert a into PriorityQueue with priority dist(a) fi
[21]     od
[22]     while PriorityQueue ≠ ∅ do
[23]       Select and remove from PriorityQueue a vertex a with minimum priority value
[24]       for every vertex b such that a → b ∈ E(G) and length(a → b) + dist(b) = dist(a) do
[25]         Insert edge a → b into SP(G)
[26]       od
[27]       for every vertex c such that c → a ∈ E(G) and length(c → a) + dist(a) < dist(c) do
[28]         dist(c) := length(c → a) + dist(a)
[29]         if c ∈ PriorityQueue then Decrease the priority of c in PriorityQueue to dist(c)
[30]         else Insert c into PriorityQueue with priority dist(c)
[31]         fi
[32]       od
[33]     od
[34]   fi
[35] else Remove edge v → w from E(G)
[36] fi
postconditions
    v → w ∉ E(G)
    SP(G) consists of all edges in E(G) on shortest paths to sink(G)
    ∀ v ∈ V(G), dist(v) is length of the shortest path from v to sink(G)
end
```

Figure 1. Procedure DeleteEdge$_{SSSP>0}$ updates the *dist* values of G after edge $v \longrightarrow w$ is deleted from G, and also updates $SP(G)$ to contain the edges of the new graph of shortest paths to $sink(G)$.

Dijkstra's batch algorithm to AFFECTED and its neighboring edges (*cf.*, the descriptions of Dijkstra's algorithm given in references [45] and [41]).

Phase 1: Identifying vertices in AFFECTED

As we now argue, the value of variable AffectedVertices at the end of Phase 1 of procedure DeleteEdge$_{SSSP>0}$ equals AFFECTED, the set of vertices of G whose shortest distance to $sink(G)$ has increased.

Phase 1 is based on the observation that a vertex u is a member of the set AFFECTED iff edge $v \longrightarrow w$ is part of *all* shortest paths from u to $sink(G)$ in the original graph G (*i.e.*, before edge $v \longrightarrow w$ is deleted from G). These vertices are successively identified during the loop on lines [8]–[15] and placed in the set AffectedVertices.

The invariant of the loop on lines [8]–[15] is that for every vertex u that is a member of the set AffectedVertices, edge $v \longrightarrow w$ is part of *all* shortest paths from u to $sink(G)$ in the original graph G (before the deletion of edge $v \longrightarrow w$). Initially AffectedVertices is empty, and each iteration of the loop on lines [8]–[15] places an additional vertex with this property in AffectedVertices. After edge $x \longrightarrow u$ is removed from $SP(G)$ on line [12], the number of $SP(G)$ edges out of x is then tested (see line [13]). If the number of such edges is zero then all shortest paths starting from x in the original graph immediately lead to vertices in AffectedVertices, and hence to edge $v \longrightarrow w$. Because for each $x \in$ AffectedVertices, edge $v \longrightarrow w$ is part of *all* shortest paths from x to $sink(G)$ in the original graph, in the modified graph the distance from x to $sink(G)$ is greater than $dist(x)$. Therefore, AffectedVertices $\subseteq$ AFFECTED.

We now argue in the other direction to show that AffectedVertices $\supseteq$ AFFECTED: At the end of the loop on lines [8]–[15], for each vertex $y \notin$ AffectedVertices there is still at least one path to $sink(G)$ such that all edges of the path are shortest-path edges of the original graph and none of the edges of the path are the edge $v \longrightarrow w$. Thus, the length of this path is $dist(y)$. Because the removal of edge $v \longrightarrow w$ cannot decrease the length of any path in G, $dist(y)$ still indicates the length of the shortest path from y to $sink(G)$. Thus $y \notin$ AFFECTED, and consequently AffectedVertices $\supseteq$ AFFECTED.

By the arguments given in the two previous paragraphs, we conclude that AffectedVertices = AFFECTED.

We now turn to the time complexity of Phase 1. During the loop on lines [8]–[15], each vertex u that is a member of AFFECTED is processed exactly once; u is placed in the workset after the number of $SP(G)$ edges out of u drops to zero. When vertex u is processed, each $SP(G)$ edge that enters u is examined during the loop on lines [11]–[14].

We would like for the processing of u to involve only a constant amount of work. Because each vertex x that is a predecessor of u via an SP edge is not necessarily a member of AFFECTED, care must be taken with the implementation of the conditional statement on line [13]:

if there does not exist a vertex y such that $x \longrightarrow y \in SP(G)$ **then** . . . **fi**

Although vertex x is guaranteed to be a *predecessor* of a member of AFFECTED (and hence at most one "step" away from AFFECTED), all that can be said about vertex y is that y is a *successor of a predecessor* of a member of AFFECTED (and thus y may be as far as *two* steps away from AFFECTED). However, by maintaining at each vertex x a count of the number of SP edges that emanate from x (*i.e.*, $|\{x \longrightarrow y \mid x \longrightarrow y \in SP(G)\}|$), it is possible to perform the test on line [13] in unit time. (Similarly, it is possible to perform the test on line [3] in unit time.) These counts must be adjusted whenever DeleteEdge$_{SSSP>0}$ or InsertEdge$_{SSSP>0}$ deletes an edge from $SP(G)$ or inserts an edge into $SP(G)$. By implementing $SP(G)$ with a bit on each edge of G, it is always possible to adjust the count at the source of the edge in unit time.

During Phase 1 of DeleteEdge$_{SSSP>0}$, the target of every edge considered is in AFFECTED. A constant amount of work is performed for each edge of the subgraph induced by AFFECTED together with all SP edges outside AFFECTED whose target is a vertex of AFFECTED. Thus, a bound on the running time of

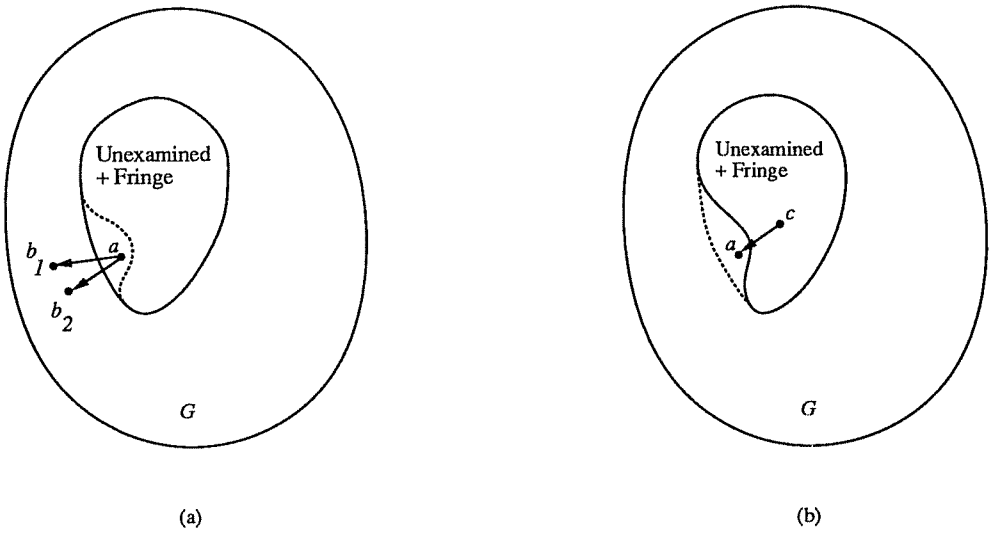
Phase 1 is $O(\| \text{AFFECTED} \|)$.

Phase 2: Determining new distances for vertices in AFFECTED and updating SP (G)

As stated earlier, Phase 2 of DeleteEdge$_{SSSP>0}$ is an adaptation of Dijkstra's batch shortest-path algorithm that uses priority-first search [41]. Because all members of AFFECTED have been identified during Phase 1, it is possible (in a very rough sense) to "pick up the process of assigning *dist* values in the middle" simply by initializing PriorityQueue with all edges $a \longrightarrow b$ where $a \in$ AFFECTED but $b \notin$ AFFECTED. (See lines [17]–[21].)

Phase 2 can be most easily understood by dividing the members of AFFECTED into three sets: *consistent vertices* (= those members of AFFECTED that have been removed from PriorityQueue during Phase 2), whose distances to *sink* (G) are correct; *fringe vertices* (= those vertices in PriorityQueue), which can have incorrect *dist* values but are queued for later processing; and *unexamined vertices* (= all other members of AFFECTED), which have yet to be considered during the loop on lines [22]–[33].

The core processing step of Phase 2 is to move from the fringe the vertex a whose distance to *sink* (G) is least among all fringe vertices, as shown below:



As part of the processing step, the current *dist* value of each predecessor c of a is examined to see whether it can be reduced in value (see lines [27]–[32]). Because vertices whose *dist* value equals ∞ are not queued in PriorityQueue, when *dist* (c) has been reduced from ∞ to some finite value it is necessary to insert c into PriorityQueue (see line [30]).

In addition to the various adjustments of *dist* values that are performed, the basic processing step also performs a loop over a's successors (see lines [24]–[26]). During this loop all edges $a \longrightarrow b$ along which there is a least-cost path to *sink* (G)—which is determined by testing whether *length* $(a \longrightarrow b)$ + *dist* (b) = *dist* (a)—are inserted into *SP* (G). This loop is performed because the goal of DeleteEdge$_{SSSP>0}$ is to update *SP* (G)—the edges of the *DAG* of shortest paths to *sink* (G)—rather than just building a shortest-path *tree* (*cf.*, the descriptions of Dijkstra's algorithm given in references [45] and [41]).

Do not be mislead by the rough characterization of Phase 2 that we gave at the beginning of the section (namely, that Phase 2 "picks up the process of assigning *dist* values in the middle") into thinking that DeleteEdge$_{SSSP>0}$ simply "rolls back" Dijkstra's batch shortest-path algorithm to the state just before vertex v was moved from the fringe, and then re-executes the batch algorithm to completion. The key point is

that the only vertices that have to be re-processed are ones in AFFECTED, which were all identified during Phase 1. Consequently, a more accurate characterization of Phase 2 is that Phase 2 is the application of Dijkstra's batch algorithm to AFFECTED and its neighoring edges.

During Phase 2 it is not necessary to consider any edge $s \longrightarrow t$ where $s \notin$ AFFECTED but $t \in$ AFFECTED. If any such edge $s \longrightarrow t$ was formerly part of a shortest path from s to $sink(G)$, $s \longrightarrow t$ is removed from $SP(G)$ during Phase 1 of DeleteEdge$_{SSSP>0}$ (see line [12]). By definition, the deletion of edge $v \longrightarrow w$ increases the distance to $sink(G)$ for all vertices in AFFECTED (including vertex t) but does not increase the distance to $sink(G)$ for vertices not in AFFECTED (such as vertex s); thus, it is not possible for edge $s \longrightarrow t$ to be part of a shortest path from s to $sink(G)$.

At the end of Phase 2, the following two conditions again hold:

(1) $SP(G)$ consists of all edges in $E(G)$ on shortest paths to $sink(G)$.
(2) For all $v \in V(G)$, $dist(v)$ is length of the shortest path from v to $sink(G)$.

During Phase 2, every edge considered has at least one endpoint in AFFECTED, and the PriorityQueue has at most $O(|\text{AFFECTED}|)$ members. Therefore, the running time for each priority-queue operation takes at most $O(\log |\text{AFFECTED}|)$, and a bound on the running time of Phase 2 is $O(\|\text{AFFECTED}\| \log |\text{AFFECTED}|)$.

Because the running time of Phase 1 is bounded by $O(\|\text{AFFECTED}\|)$ and the running time of Phase 2 is bounded by $O(\|\text{AFFECTED}\| \log |\text{AFFECTED}|)$, the total running time of DeleteEdge$_{SSSP>0}$ is bounded by $O(\|\delta\| \log \|\delta\|)$.

3.1.2. Insertion of an Edge

We now turn to the problem of how to update distances (and also the set $SP(G)$) after an edge $v \longrightarrow w$ with length c is inserted into G. The algorithm for this problem, procedure InsertEdge$_{SSSP>0}$, is presented in Figure 2.

The idea behind InsertEdge$_{SSSP>0}$ is as follows. 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 $sink(G)$. Furthermore, if u is an affected vertex and x is the successor of u in the shortest path from u to v, then x must itself be an affected vertex. This implies that the necessary updating could be done easily *if only we had available T, a single-sink shortest-path tree for the vertex v*:

> If a single-sink shortest-path tree T were available for vertex v, we could perform a traversal of the tree T, starting from the root v. Whenever we visited a vertex u, we would check to see if u were an affected vertex: u is affected iff $dist(u,v) + length(v \longrightarrow w) + dist(w) < dist(u)$, where $dist(u,v)$ is the length of the shortest path from u to v (this information would be available from the tree T) and $dist(u)$ and $dist(w)$ are the lengths of the previous shortest paths from u and w to $sink(G)$, respectively. If u was found to be affected, we would update $dist(u)$ to $dist(u,v) + length(v \longrightarrow w) + dist(w)$ and continue the tree traversal by visiting the children of u in T. If u was found not to be affected, then none of its descendents in T could be affected, and hence there would be no need to visit them during the tree traversal. This would guarantee that all the visited vertices were either affected, or were predecessors in G of affected vertices, and hence the algorithm would be a bounded one.
>
> The algorithm would also have to update the graph $SP(G)$. Note that for every edge $x \longrightarrow y$ that must be deleted from $SP(G)$, x must be an affected vertex; similarly, for every edge $x \longrightarrow y$ that must be added to $SP(G)$, y must be an affected vertex. Thus, updating $SP(G)$ could be done as the affected vertices are identified.

In order to carry out the steps described above—but without constructing tree T—InsertEdge$_{SSSP>0}$ adapts Dijkstra's algorithm to identify shortest paths to v, but only examines affected vertices and their neighbors,

```
procedure InsertEdge_SSSP>0(G, v → w, c)
declare
    G: a directed graph
    v → w: an edge to be inserted in G
    c: a positive real number indicating the length of edge v → w
    PriorityQueue: a priority queue of vertices
    u, x: vertices
preconditions
    v → w ∉ E(G)
    SP(G) consists of all edges in E(G) on shortest paths to sink(G)
    ∀ v ∈ V(G), dist(v) is length of the shortest path from v to sink(G)
begin
[1]  Insert edge v → w into E(G)
[2]  length(v → w) := c
[3]  PriorityQueue := ∅
[4]  if length(v → w) + dist(w) < dist(v) then
[5]     dist(v) := length(v → w) + dist(w)
[6]     Insert v into PriorityQueue with priority 0
[7]  else if length(v ⇢ w) + dist(w) = dist(v) then
[8]     Insert v → w into SP(G)
[9]  fi
[10] while PriorityQueue ≠ ∅ do
[11]    Select and remove a vertex u in PriorityQueue with minimum priority
[12]    Remove all edges of SP(G) directed away from u
[13]    for every vertex x such that u → x ∈ E(G) do
[14]       if length(u → x) + dist(x) = dist(u) then Insert u → x into SP(G) fi
[15]    od
[16]    for every vertex x such that x → u ∈ E(G) do
[17]       if length(x → u) + dist(u) < dist(x) then
[18]          dist(x) := length(x → u) + dist(u)
[19]          if x ∈ PriorityQueue then
[20]             Decrease the priority of x in PriorityQueue to dist(x) – dist(v)
[21]          else
[22]             Insert x into PriorityQueue with priority dist(x) – dist(v)
[23]          fi
[24]       else if length(x → u) + dist(u) = dist(x) then
[25]          Insert x → u into SP(G)
[26]       fi
[27]    od
[28] od
postconditions
    v → w ∈ E(G)
    length(v → w) = c
    SP(G) consists of all edges in E(G) on shortest paths to sink(G)
    ∀ v ∈ V(G), dist(v) is length of the shortest path from v to sink(G)
end
```

Figure 2. Procedure InsertEdge$_{SSSP>0}$ updates the *dist* values of G after edge $v \longrightarrow w$ is inserted into G (with $length(v \longrightarrow w) = c$). InsertEdge$_{SSSP>0}$ also updates $SP(G)$ to contain the edges of the new graph of shortest paths to $sink(G)$.

which ensures that InsertEdge$_{SSSP>0}$ is bounded.

Note that unlike procedure DeleteEdge$_{SSSP>0}$, 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$_{SSSP>0}$ the identification of AFFECTED is *interleaved* with updating.

We now consider the time complexity of InsertEdge$_{SSSP>0}$. During InsertEdge$_{SSSP>0}$, every edge considered (except possibly the inserted edge $v \longrightarrow w$) has at least one endpoint in AFFECTED. The PriorityQueue has at most $O(|\text{AFFECTED}|)$ members, and the number of PriorityQueue operations performed is bounded by $\|\text{AFFECTED}\|$, the number of edges considered. Thus, a bound on the running time of procedure InsertEdge$_{SSSP>0}$ is $O(\|\text{AFFECTED}\| \log |\text{AFFECTED}|)$, and hence $O(\|\delta\| \log \|\delta\|)$, where the log factor is due to the required manipulations of PriorityQueue.

3.1.3. Incremental Updating in the Presence of Negative Edge Lengths

We now show that the operations DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$ can be generalized to handle non-positive edge lengths. Batch algorithms for identifying shortest paths in graphs with arbitrary edge lengths normally assume that the sum of the lengths of the edges in any cycle is greater than or equal to 0. In contrast, we will restrict our attention to graphs in which the sum of the lengths of the edges in any cycle is strictly greater than 0. (We show in Section 4.2 that there exists *no* bounded locally persistent incremental algorithm for maintaining shortest paths if 0-weight cycles are allowed.)

The versions of DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$ given in Sections 3.1.1 and 3.1.2 cannot handle 0-weight cycles because the *SP* graph need not be a DAG in the presence of 0-weight cycles: every edge in a 0-weight cycle will be in the *SP* graph, provided the sink is reachable from the cycle. However, DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$ *can* handle 0-weight *edges* as long as there are no 0-weight *cycles*. DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$ cannot handle negative-length edges for the same reasons that Dijkstra's algorithm cannot.

We can generalize DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$ to handle negative-length edges (as long as all cycles have net positive weight) by adapting the technique of Edmonds and Karp for transforming the length of every edge to a non-negative real without changing the graph's shortest paths [12, 45]. Their technique is based on the observation that if f is any function that maps vertices of the graph to reals, and the length of each edge $a \longrightarrow b$ is replaced by $f(b) + length(a \longrightarrow b) - f(a)$, then the shortest paths in the graph are unchanged from the original edge-length mapping. If f satisfies the property that $f(b) + length(a \longrightarrow b) - f(a) \geq 0$ for every edge $a \longrightarrow b$ in the graph, then Dijkstra's algorithm can be applied.

We note that for the problem of incrementally updating shortest paths to a sink such a function f is available; we simply let $f(u)$ be $dist_{old}(u)$, the length of the shortest path in graph G from u to $sink(G)$ before G was modified. Note that for every edge $a \longrightarrow b$ in the original graph $dist_{old}(a) \leq dist_{old}(b) + length(a \longrightarrow b)$, and hence, with one slight exception, $dist_{old}$ meets the properties required of f: in the case of an edge insertion, the transformed length of the newly inserted edge $v \longrightarrow w$ is not guaranteed to be non-negative; however, this causes no difficulties because, as explained in Section 3.1.2, InsertEdge$_{SSSP>0}$ uses an adaptation of Dijkstra's shortest-path algorithm to identify shortest paths to vertex v, none of which can include the edge $v \longrightarrow w$.

Because $dist_{old}(u)$ is already available, it is necessary to make only slight adaptations of the procedures DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$ given in Sections 3.1.1 and 3.1.2 to make use of the transformed lengths. This does not change the complexities of the update procedures (*i.e.*, updating can still be carried out in time bounded by $\|\delta\|$). We leave the details to the reader.

3.2. The Incremental All-Pairs Shortest-Path Problem

This section concerns a bounded incremental algorithm for a version of the incremental all-pairs shortest-path problem with positive edge weights (APSP>0). We deal only with the restricted version of the problem in which the set of vertices of the underlying graph G is static; that is, it is not permitted to add or remove vertices from G. (A technique for eliminating this restriction will be discussed shortly in footnote

6.) We will assume that the vertices of G are indexed from $1..|V(G)|$. As in Section 3.1, we describe only the operations that update distances when edges are added or removed from the graph; it is trivial to generalize these operations to handle the shortening or lengthening of an edge, respectively.

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. One can also 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.[6] (When this scheme is used, we say that the distance matrix is *stored at sources*.) Thus, APSP>0 does not fall into the class of graph problems that involve the computation of a *single* value $S_G(u)$ for each vertex u of the input graph G, and so, as explained below, some of our terminology in this section differs from the terminology that was introduced in Section 2.

Whereas the definition of MODIFIED remains the same (and hence for a single-edge change to the graph $|\text{MODIFIED}| = 2$), AFFECTED is defined to be the set of entries in distance matrix *dist* that require new values after an edge-modification to G, and hence

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

where AFFECTED_u represents the set of affected vertices for the single-sink problem with sink vertex u. Note that for a given change δ, some or all of the AFFECTED_u can be empty. 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 definitions of AFFECTED, $\|\text{AFFECTED}\|_i$, and $\|\delta\|_i$ given above are clearly in the same spirit as those from Section 2. As we shall see, $\|\delta\|_2$ is the appropriate parameter in which to measure the incremental complexity of $\text{DeleteEdge}_{APSP>0}$, whereas $\|\delta\|_1$ is the appropriate parameter in which to measure the incremental complexity of $\text{InsertEdge}_{APSP>0}$.

Aside. One of the reasons that we have introduced the idea of having the distance matrix stored at sources is that the concept of a global auxiliary distance matrix is technically outside the model of *locally persistent algorithms*, the model of incremental computation that is introduced in Section 4 for studying lower bounds on incremental-computation problems. The device of storing the distance matrix at sources allows the results from Section 3.2 to be compatible with those of Section 4.
End Aside.

3.2.1. Deletion of an Edge

The basic idea behind the bounded incremental algorithm for $\text{DeleteEdge}_{APSP>0}$ is to make repeated use of the bounded incremental algorithm $\text{DeleteEdge}_{SSSP>0}$ as a subroutine, but with a different sink vertex on each call. A simple incremental algorithm for $\text{DeleteEdge}_{APSP>0}$ would be to make as many calls on

---

[6]The operations of inserting and deleting (isolated) vertices would introduce some additional concerns having to do with dynamic storage allocation. When the distance matrix is distributed over a collection of arrays in the fashion described above, we note that by dynamically expanding and contracting these arrays using the well-known doubling/halving technique employed in expanding/contracting (or "breathing") hash tables, it is possible to perform an insertion or deletion of an isolated vertex with an *amortized* cost of $O(|V(G)|)$: doubling or halving all the arrays in the graph takes time $O(|V(G)|^2)$, but the cost is amortized over $\Omega(|V(G)|)$ 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.

DeleteEdge$_{SSSP>0}$ 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 $O(|V(G)|)$, 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 DeleteEdge$_{APSP>0}$ is that it is possible to determine *exactly* which calls on DeleteEdge$_{SSSP>0}$ 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 DeleteEdge$_{SSSP>0}$ as stated in Figure 1 actually performs the deletion of edge $v \longrightarrow w$ from graph G (see lines [2] and [35]), a few changes in DeleteEdge$_{SSSP>0}$ are necessary for it to be called multiple times in the manner suggested above.

There is also a more serious problem with using procedure DeleteEdge$_{SSSP>0}$ from Figure 1 in conjunction with the ideas outlined above. The problem is that DeleteEdge$_{SSSP>0}$ 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

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

$$= \|\text{MODIFIED}\|_2.$$

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$_{SSSP>0}$ from Figure 1 as a subroutine in DeleteEdge$_{APSP>0}$ 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$_{APSP>0}$. Procedure DeleteUpdate is presented in Figure 3. DeleteUpdate is very similar to DeleteEdge$_{SSSP>0}$, but eliminates the two problems discussed above. DeleteUpdate does not delete any edges; the deletion of edge $v \longrightarrow w$ is performed in DeleteEdge$_{APSP>0}$ itself (see line [1] of Figure 4). In addition, DeleteUpdate does not need to update any SP information explicitly, because SP 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?"

The use of predicate $SP(a, b, c)$ has some consequences with regard to the time bound on procedure DeleteUpdate (and hence on DeleteEdge$_{APSP>0}$). In particular, compare the test on line [10] of Figure 3 with the one on line [13] of Figure 1. In the latter case, it is possible for the test to be performed in unit time by maintaining at vertex x a count of the number of SP edges that emanate from x; the count would have to be adjusted in DeleteEdge$_{SSSP>0}$ and InsertEdge$_{SSSP>0}$ whenever edges are inserted into or deleted from $SP(G)$. In contrast, the test on line [10] of Figure 3 must be performed by explicitly testing at least some of the successors of x to see whether the conditions $SP(x, y, z)$ and $y \in$ AffectedVertices hold. Note that vertex x is not necessarily a member of AFFECTED, but when x is not a member of AFFECTED it is guaranteed to be a predecessor of a member of AFFECTED. Thus, vertex y is no more than *two* steps beyond AFFECTED.

Because of the way SP information is handled in Phase 1 of DeleteUpdate, the cost of Phase 1 is bounded by $O(\|\delta\|_2)$; Phase 2 is bounded by $O(\|\text{AFFECTED}_z\|_1 \log |\text{AFFECTED}_z|)$. Thus the total cost of DeleteUpdate is bounded by a function of the parameter $\|\delta\|_2$ (but the cost is not bounded by a

```
procedure DeleteUpdate(G, v → w, z)
declare
    G: a directed graph
    v → 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 priority queue of vertices
    SP(a, b, c) ≡ (dist_G(a, c) = length_G(a → b) + dist_G(b, c)) ∧ (dist_G(a, c) ≠ ∞)
begin
[1]  AffectedVertices := ∅
[2]  if there does not exist any vertex x ∈ succ_G(v) such that SP(v, x, z) then
[3]   /* Phase 1: Identify vertices in AFFECTED (the set of vertices whose shortest distance to z has increased). */
[4]   /*          Set AffectedVertices equal to AFFECTED. */
[5]    WorkSet := { v }
[6]    while WorkSet ≠ ∅ do
[7]      Select and remove a vertex u from WorkSet
[8]      Insert vertex u into AffectedVertices
[9]      for each vertex x ∈ pred_G(u) such that SP(x, u, z) do
[10]       if for all vertices y ∈ succ_G(x) such that SP(x, y, z), y ∈ AffectedVertices then Insert vertex x into WorkSet fi
[11]     od
[12]   od
[13]  /* Phase 2: Determine new distances to z for all vertices in AffectedVertices. */
[14]   PriorityQueue := ∅
[15]   for each vertex a ∈ AffectedVertices do
[16]     dist_G(a, z) := min ({ length_G(a → b) + dist_G(b, z) | a → b ∈ E(G) and b ∈ (V(G) − AffectedVertices) } ∪ { ∞ })
[17]     if dist_G(a, z) ≠ ∞ then Insert a into PriorityQueue with priority dist_G(a, z) fi
[18]   od
[19]   while PriorityQueue ≠ ∅ do
[20]     Select and remove from PriorityQueue a vertex a with minimum priority value
[21]     for every vertex c such that c → a ∈ E(G) and length_G(c → a) + dist_G(a, z) < dist_G(c, z) do
[22]       dist_G(c, z) := length_G(c → a) + dist_G(a, z)
[23]       if c ∈ PriorityQueue then Decrease the priority of c in PriorityQueue to dist_G(c, z)
[24]       else Insert c into PriorityQueue with priority dist_G(c, z)
[25]       fi
[26]     od
[27]   od
[28] fi
end
```

Figure 3. Procedure DeleteUpdate updates distances to vertex z after edge $v \longrightarrow w$ is deleted from G.

function of $\|\delta\|_1$, unlike the cost of procedure DeleteEdge$_{SSSP>0}$ from Figure 1[7]). As a consequence, the cost of DeleteEdge$_{APSP>0}$ is also bounded by a function of $\|\delta\|_2$, but not by a function of the parameter $\|\delta\|_1$. (This is the reason for our earlier remark that $\|\delta\|_2$, rather than $\|\delta\|_1$, is the appropriate parameter in which to measure the incremental complexity of DeleteEdge$_{APSP>0}$.)

Procedure DeleteEdge$_{APSP>0}$ is given in Figure 4. Procedure DeleteEdge$_{APSP>0}$ 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 $dist_G(u, v)$

[7]Clearly, DeleteEdge$_{SSSP>0}$ could be modified to use the *SP* predicate instead of maintaining *SP* edges. In this case, the cost of DeleteEdge$_{SSSP>0}$ would be $O(\|\delta\|_2 + \|\delta\|_1 \log \|\delta\|_1)$. Thus, we see that maintaining the *SP* graph of shortest paths enables DeleteEdge$_{SSSP>0}$ to run in time that depends on a lower-order extended size of CHANGED.

```
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 4. Procedure DeleteEdge$_{APSP>0}$ updates distances 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 $dist_{\bar{G}}(u, v)$ is the length of the shortest path from v to u in G. Note that a single-*sink* problem in graph $\bar{G}$ is equivalent to a single-*source* shortest-path problem in graph G. Thus, we will henceforth speak in terms of "solving single-source problems" synonymously with "solving single-sink problems in $\bar{G}$."

Both of these graphs are updated, as described earlier, by performing 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 $\bar{G}$. 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 $\bar{G}$. 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($\bar{G}$, $w \longrightarrow v$, 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($\bar{G}$, $w \longrightarrow v$, 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$_{APSP>0}$ is bounded by

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

which in turn is bounded by $O(\|\delta\|_2 + \|\delta\|_1 \log \|\delta\|_1)$. (Bear in mind that, in general, some of the AFFECTED$_u$ are empty.)

3.2.2. Insertion of an Edge

We now turn to the problem of how to update distances after an edge $v \longrightarrow w$ of length c is inserted into G. 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 $\text{InsertEdge}_{SSSP>0}$ with y as the sink. The dual information maintained in $\overline{G}$ is updated in an identical fashion.

The algorithm $\text{InsertEdge}_{APSP>0}$ presented in Figure 6 carries out essentially the technique outlined above, but with one difference. It makes use of a considerably simplified form of the procedure $\text{InsertEdge}_{SSSP>0}$, which is given as procedure InsertUpdate in Figure 5. The simplifications incorporated in InsertUpdate are explained below.

Recall the description of $\text{InsertEdge}_{SSSP>0}$ given in Section 3.1.2. $\text{InsertEdge}_{SSSP>0}$ 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 $\text{InsertEdge}_{APSP>0}$.) 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

---

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 :=∅
[4] **while** WorkSet ≠ ∅ **do**
[5] Select and remove an edge $x \longrightarrow y$ from WorkSet
[6] **if** $length_G(x \longrightarrow y) + dist_G(y,z) < dist_G(x,z)$ **then**
[7] Insert x into AffectedVertices
[8] $dist_G(x,z) := length_G(x \longrightarrow y) + dist_G(y,z)$
[9] **for** every vertex u such that $u \longrightarrow x \in E(G)$ **do**
[10] **if** $SP(u,x,v)$ **and** $u \notin$ VisitedVertices **then**
[11] Insert $u \longrightarrow x$ into WorkSet
[12] Insert u into VisitedVertices
[13] **fi**
[14] **od**
[15] **fi**
[16] **od**
end

Figure 5. Procedure InsertUpdate updates distances to vertex z after edge $v \longrightarrow w$ is inserted into G.

traversal of this DAG, visiting only affected vertices or their predecessors.

For instance, consider the edge $x \longrightarrow y$ selected in line [5] of Figure 5. Vertex x is the vertex to be visited next during the traversal described above. Except in the case when edge $x \longrightarrow y$ is $v \longrightarrow w$, vertex y 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$_{APSP>0}$ 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$_{APSP>0}$ must invoke InsertUpdate. Assume that x and y are 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 | 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$_{APSP>0}$ determines the set AffectedSinks (see line [5] of Figure 6); 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$_{APSP>0}$. 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(AFFECTED$_x$)>, in time $O(\| \text{AFFECTED}_x \|)$. Thus, the total running time of line [7] in procedure InsertEdge$_{APSP>0}$ 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$_{APSP>0}$ is $O(\| \delta \|_1)$.

---

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 $\overline{E}(G)$
[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 6. Procedure InsertEdge$_{APSP>0}$ updates distances after edge $v \longrightarrow w$ of length c is inserted in G.

3.3. The Incremental Circuit-Value Problem

A *circuit* is a DAG where 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 evaluation of each function F_u is assumed to take unit time. (This implies that the indegrees of all vertices in the circuit are bounded by some constant.) The circuit-value problem is to compute the output value associated with each vertex. Alpern *et al.* show that the incremental circuit-value 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-value problem that runs in time $O(2^{2\times\|\delta\|})$. Previous to our work, no bounded algorithm for the incremental circuit-value 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 incremental circuit-value 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. We call u the *modified* vertex.

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 incremental circuit-value 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 incremental circuit-value problem.

The algorithm outlined here does not maintain any topological ordering of the DAG. Consequently, the algorithm might visit vertices out of order while propagating changes. This can cause vertices to be assigned a wrong value temporarily. Eventually all vertices will be assigned the correct values. If an unaffected vertex v is assigned a wrong value (temporarily), this can cause changes to propagate spuriously beyond v. For the incremental algorithm to be bounded, it is necessary that the number of vertices visited during change propagation be bounded (by a function of $\|\delta\|$). We will subsequently see how, by carefully choosing the order in which vertices are visited, the propagation of spurious changes can be kept within bounds.

We first outline an algorithm schema for propagating changes and prove its correctness. The algorithm schema is given in Figure 7, and the idea behind it is the following: The algorithm builds a set WorkSet

that is a conservative approximation to AFFECTED: eventually WorkSet will contain all affected vertices. The WorkSet initially consists of only the modified vertex u, and is iteratively expanded by choosing a set of vertices ToBeExpanded (see line [7]) and adding all the successors of vertices in ToBeExpanded to WorkSet (lines [8]–[14]). Vertices in ToBeExpanded are said to have been *expanded* at this stage. After each such expansion, values of vertices in WorkSet are recomputed in a *relative topological-sort* order. Let H denote the subgraph of G induced by the set of vertices WorkSet. Any topological sorting of H yields a *relative topological-sort* order for WorkSet. Note in particular that if a vertex u topologically precedes vertex v in G, but all paths in G from u to v pass through some vertex not in WorkSet, then u need not come before v in a relative topological-sort order for WorkSet. This is important because it may not be possible to determine an actual topological-sort ordering of the vertices of WorkSet (*i.e.*, an ordering that accounts for all paths in G) in time bounded by a function of |WorkSet| (or even $\|\text{WorkSet}\|_i$ for any fixed value of i). In contrast, under the assumption that each vertex has a bounded number of predecessors, it is possible to determine a relative topological-sort ordering of the vertices of WorkSet in time $O(|\text{WorkSet}|)$.

The unexpanded vertices in WorkSet are called the *inner fringe* vertices. Vertices outside WorkSet that have some predecessor in WorkSet are called *outer fringe* vertices. The algorithm halts when the recomputed value of every inner fringe vertex is the same as the original value of the vertex.

```
procedure UpdateSchema (G, u)
declare
    G : an annotated circuit
    u : a vertex in G
    WorkSet, InnerFringe, ToBeExpanded : sets of vertices
    v, w: vertices
preconditions
    Every vertex in G except possibly u is consistent
begin
[1]  WorkSet := { u }
[2]  InnerFringe := { u }
[3]  u.originalValue := u.value
[4]  loop
[5]    for every vertex v ∈ WorkSet in relative topological-sort order do recompute v.value od
[6]    if for all v ∈ InnerFringe, v.value = v.originalValue then exit loop fi
[7]    Choose some non-empty subset ToBeExpanded of InnerFringe
[8]    for every vertex v ∈ ToBeExpanded do
[9]      Remove v from InnerFringe
[10]     for every vertex w ∈ (succ(v) – WorkSet) do
[11]       Insert w into both WorkSet and InnerFringe
[12]       w.originalValue := w.value
[13]     od
[14]   od
[15] end loop
postconditions
    Every vertex in G is consistent
end
```

Figure 7. A program schema for the incremental circuit-value problem.

Proposition 3.1. *Procedure UpdateSchema 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, assuming the inductive hypothesis that every predecessor of v in G is correct, 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. (Whenever *v.value* is recomputed, v becomes consistent. It can subsequently become inconsistent only if some predecessor of v has its value recomputed.) It follows that vertex v is also correct (since its predecessors are correct, according to the inductive hypothesis).

Now consider the case that v is not in WorkSet. Here, v and all of its predecessors have the same values as they did before the update. (Any predecessor w of v that is in WorkSet must have been an unexpanded vertex, and hence, an inner fringe vertex. Since when the algorithm terminates every inner fringe vertex has the same value as it did originally, it follows that w has the same value as it did originally. Any predecessor of v that is not in WorkSet obviously has the same value as it did initially.) Since v was initially consistent (from the precondition of the procedure), it must still be consistent and, hence, correct. It follows that *UpdateSchema* computes a correct annotation of the circuit. □

The set of vertices to be expanded are chosen (nondeterministically) at line [7] of *UpdateSchema*. If these vertices are chosen arbitrarily, the resulting algorithm will be an unbounded one. The problem is that an unbounded amount of propagation of spurious changes is possible. A simple scheme that keeps the algorithm bounded in the case of DAGs of bounded outdegree is to propagate changes in a breadth-first order. We achieve this by expanding all inner fringe vertices during each iteration. Let *SimpleUpdate* be the procedure obtained from *UpdateSchema* by refining line [7] to "ToBeExpanded := InnerFringe" .

Proposition 3.2. *Procedure SimpleUpdate computes the correct annotation of circuit G in time* $O(k^{|\text{AFFECTED}|})$, *where each vertex in G has outdegree* $\leq k$.

Proof. The proof that the computed annotation is correct follows from Proposition 3.1. 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 all of the vertices that are added to WorkSet in lines [8]–[14] in a particular iteration, say the i-th one, are 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, assuming the inductive hypothesis that every predecessor of v in G is correct, 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 (since its predecessors are correct, according to the inductive hypothesis).

Now consider the case that v is not in WorkSet. If v is an outer fringe vertex (*i.e.*, one of the vertices added to WorkSet in lines [8]–[14]), then v is an unaffected vertex according to the above hypothesis, and hence has the correct value (since the correct value for an unaffected vertex is its original value).

If v is neither an outer fringe vertex nor a WorkSet vertex, then it must have the same value as it did initially. Therefore v is correct because all its predecessors also have the same values as they did initially.

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), and the algorithm halts after the i+1-th iteration. It follows from the above argument that the algorithm makes at most |AFFECTED|+1 iterations.

Every vertex in the circuit has outdegree at most k. It can be verified by induction that at the beginning of the i-th iteration $|\text{InnerFringe}| \leq k^{i-1}$. Hence, at most k^i new vertices can be added to WorkSet during the i-th iteration. Hence, at the beginning of the i-th iteration, $|\text{WorkSet}| \leq \sum_{j=0}^{i-1} k^j = (k^i-1)/(k-1)$. The i-th iteration itself takes time $k \times (k^i-1)/(k-1)$. The whole algorithm takes time

$$O\Big(\sum_{i=1}^{|\text{AFFECTED}+1|} k \times (k^i-1)/(k-1)\Big) = O(k^{|\text{AFFECTED}|}). \ \square$$

Note that *SimpleUpdate* can be unbounded, if we do not assume that the outdegree of every vertex is bounded by k. With procedure *SimpleUpdate*, 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 *SimpleUpdate* 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 now outline a refinement of the procedure *UpdateSchema* that leads to an $O(2^{2\times\|\delta\|})$ algorithm. We do not require the vertices to have bounded outdegree, though we still assume that the indegrees of vertices are bounded. The algorithm makes use of the outdegree of the vertices, which can be maintained trivially. Let *u.outdegree* denote the outdegree of vertex u.

Whereas *SimpleUpdate* expands all the inner fringe vertices in each iteration, the algorithm outlined in Figure 8 expands exactly one carefully chosen inner fringe vertex. The chosen vertex v is such that adding v's successors to WorkSet will not cause the algorithm to be unbounded. How is this vertex v chosen?

Consider a path P from vertex x to vertex y. Assume that all the vertices in this path have been expanded. The sum of the outdegrees of the expanded vertices is an upper bound on the number of vertices added to WorkSet during these expansions. If every vertex in the path were an affected vertex, then the sum of the outdegrees of the vertices in the path is a lower bound for $\|\text{AFFECTED}\|$. Given a path P from x to y in a directed graph F, let $cost_F(P)$ be the sum of the outdegrees of all vertices except y in path P. For any two vertices x and y in graph F, let $cost_F(x,y)$ be the minimum over all paths P from x to y of $cost_F(P)$. Procedure *Update* is very much like Dijkstra's single-source shortest-path algorithm where the *length* function on the edges of G is $length(a \longrightarrow b) = a.outdegree$. It computes $v.cost = cost_G(u,v)$, for every vertex v in WorkSet (but differs from Dijkstra's algorithm in that it does not compute $cost_G(u,v)$ for any vertex v outside WorkSet). In every iteration, it expands an inner fringe vertex v for which $cost_G(u,v)+v.outdegree$ is minimum, thus moving outer fringe vertices w for which $cost_G(u,w)$ is minimum to WorkSet. This, as will be shown soon, guarantees that the algorithm is bounded.

Proposition 3.3. *Procedure Update computes a correct annotation of G in time* $O(2^{2\times\|\delta\|})$.

Proof. The proof that the computed annotation is correct follows from Proposition 3.1. The proof of the time complexity follows.

In what follows, $\overline{\text{WorkSet}}$ denotes the final value of the variable WorkSet. We first show that the cardinality of $\overline{\text{WorkSet}}$ is bounded by a function of $\|\delta\|$. Assume that at least one vertex is expanded during the call on *Update* (the proof is trivial otherwise). Note that once WorkSet contains all affected vertices, recomputing values of vertices in WorkSet in a relative topological-sort order results in a correct annotation of the circuit. Then the algorithm will continue iterating until all affected vertices have been expanded. Once every affected vertex has been expanded, the algorithm will halt. This follows since at this point every inner fringe vertex (which, by definition, is an unexpanded vertex) is an unaffected vertex, and hence has the same value as it did originally (since the circuit annotation is correct). Consequently, the last vertex to be expanded, denoted by $\overline{v}$, must be an affected vertex. Hence, there must be a path P consisting only of affected vertices from u to $\overline{v}$. It follows that $\overline{v}.cost+\overline{v}.outdegree \leq cost_G(P)+\overline{v}.outdegree \leq \|\text{AFFECTED}\| \leq \|\delta\|$.

```
procedure Update (G, u)
declare
    G : an annotated circuit
    u : a vertex in G
    WorkSet, InnerFringe : sets of vertices
    v, w: vertices
preconditions
    Every vertex in G except possibly u is consistent
begin
[1]  WorkSet := { u }
[2]  InnerFringe := { u }
[3]  u.originalValue := u.value
[4]  u.cost := 0
[5]  loop
[6]     for every vertex v ∈ WorkSet in relative topological-sort order do recompute v.value od
[7]     if for all v ∈ InnerFringe, v.value = v.originalValue then exit loop fi
[8]     Select and remove a vertex v from InnerFringe for which v.cost+v.outdegree is minimum
[9]     for every vertex w ∈ (succ(v) – WorkSet) do
[10]       Insert w into both WorkSet and InnerFringe
[11]       w.originalValue := w.value
[12]       w.cost := v.cost + v.outdegree
[13]    od
[14] end loop
postconditions
    Every vertex in G is consistent
end
```

Figure 8. An exponentially bounded algorithm for the incremental circuit-value problem.

Every vertex y in $\overline{\text{WorkSet}}$ except u was added to WorkSet during the expansion of some vertex x. Call x the parent of y, denoted by parent(y). Consider the directed tree H, rooted at u, with $V(H) = \overline{\text{WorkSet}}$ and $E(H) = \{ parent(y) \longrightarrow y \mid y \in (\overline{\text{WorkSet}} - \{ u \}) \}$. ($H$ is a partial shortest-path tree for source u in graph G; H serves to characterize the work performed by *Update*.) Obviously, $cost_H(u,v) \le cost_G(u,v)$. Because $\overline{v}$ was the *last* vertex ever expanded, for each vertex w that was expanded, $cost_G(u,w) + w.outdegree \le cost_G(u,\overline{v}) + \overline{v}.outdegree$. It follows that for every vertex v in tree H, $cost_H(u,v) \le cost_G(u,parent(v)) + parent(v).outdegree \le cost_G(u,\overline{v}) + \overline{v}.outdegree = \overline{v}.cost + \overline{v}.outdegree \le \| \delta \|$.

A directed tree T with root r (with the edges being directed from the parent to the offspring) is said to be a k-tree if for every vertex v in the tree, $cost_T(r,v) \le k$. Let $N(k)$ denote the maximum possible number of vertices in a k-tree. A tree is a k-tree iff the root has outdegree $i \le k$, and each of the root's subtrees are $(k-i)$-trees. Hence,

$$N(0) = 1;$$
$$N(k) = 1 + \max_{1 \le i \le k}(i \times N(k-i)), \quad \text{for } k > 0.$$

It can be verified by induction that $2^{(k/2)} \le N(k) \le 2^k$. We need only the upper bound. Since for every vertex v in tree H, $cost_H(u,v) \le \| \delta \|$, H is a $\| \delta \|$-tree, and hence $|\overline{\text{WorkSet}}| \le 2^{\| \delta \|}$. The algorithm makes at most $|\overline{\text{WorkSet}}|+1$ iterations. Each iteration takes time $O(|\overline{\text{WorkSet}}|)$, under the assumption that each vertex has a bounded number of predecessors. Hence, the whole algorithm runs in time $O(2^{2 \times \| \delta \|})$. □

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. Furthermore, in the description of change-propagation algorithms given at the beginning of Section 3.3, we mentioned that only *potentially affected* vertices have their values recomputed. Thus, in the algorithms we described, only vertices whose values change need be expanded. We have avoided these improvements to keep the algorithms and their analysis simple. These changes will most likely improve the average-case performance greatly, but the worst-case complexity would still be exponential (in $\| \delta \|$).

The algorithm can also be generalized to handle "non-unit" changes (*i.e.*, input changes that modify multiple vertices and edges).
End Aside.

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 successors of 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 non-incremental 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 non-incremental for the class of locally persistent algorithms.

Throughout the section, unless explicitly noted otherwise, the term "non-incremental" is shorthand for "non-incremental for the class of locally persistent algorithms."

4.1. The Single-Source Reachability Problem is Non-Incremental

Definition 4.1. (*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 4.2. *SS-REACHABILITY is non-incremental 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 $\overline{d_G}(\delta_1,\overline{\delta_2})$ between the two modifications δ_1 and δ_2 to graph G is defined to be $d_J(\text{MODIFIED}_{G,\delta_1},\text{MODIFIED}_{G,\delta_2})$.

Consider the graph G shown in Figure 9. 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 not both of the changes δ_1 and δ_2 to G can be processed in constant time (*i.e.*, time independent of the size of graph G or the length of the dotted paths indicated in the figure) by a locally persistent algorithm.

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

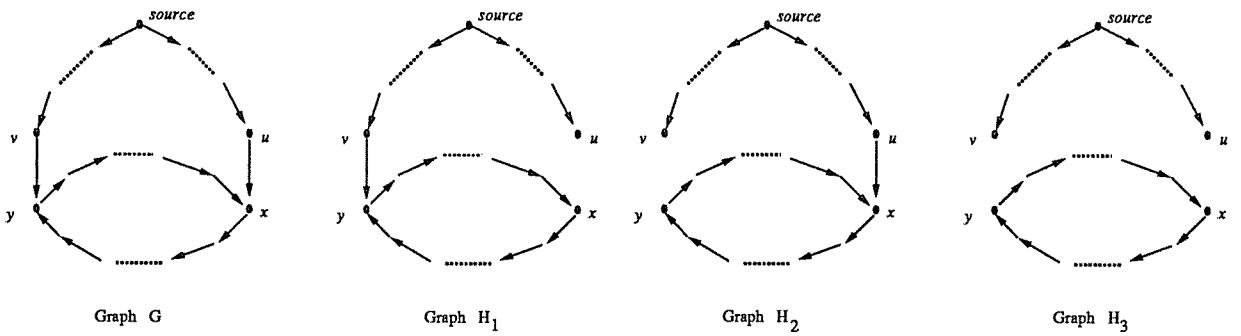


Figure 9. Graphs used in the proof that SS-REACHABILITY is non-incremental for locally persistent algorithms.

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 $\underline{G}$ 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. □.

4.2. Non-Incremental Path Problems

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

Definition 4.3. A *closed semiring* is a system $(S, \oplus, \otimes, \bar{0}, \bar{1})$ consisting of a set S, two binary operations $\oplus$ and $\otimes$ on S, and two elements $\bar{0}$ and $\bar{1}$ of S, satisfying the following axioms:

(1) $(S, \oplus, \bar{0})$ is a meet-semilattice with greatest element $\bar{0}$. (Thus, $\oplus$ is a commutative, associative, idempotent operator with identity element $\bar{0}$. 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) $(S, \otimes, \bar{1})$ is a monoid. (Thus, $\otimes$ is an associative operator with identity $\bar{1}$.)

(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) $a \otimes \bar{0} = \bar{0}$.

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

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

where $a^0 = \bar{1}$ 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 $l(p)$:

$$C(v,w) = \bigoplus_{v \rightarrow^* w} l(p).$$

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 $\mathbf{R} = (S, \oplus, \otimes, \bar{0}, \bar{1})$ be a specific closed semiring. The SS-**R** problem is defined as follows.

Definition 4.4. Given a directed graph $G = (V, E)$, a vertex s in V, and an edge-labeling function $l : E \rightarrow S$, 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 incremental 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 $(\mathcal{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0)$. Then, SS-**R** is nothing more 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 non-incremental. 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 non-incremental.

However, some caution needs to be exercised in making inferences about the non-incrementality of a problem via a reduction argument. If a problem P is non-incremental and can be reduced to a problem Q in the conventional sense, it does not necessarily follow that the problem Q is non-incremental. For instance, consider any non-incremental 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 an incremental problem.

Showing that a problem Q is non-incremental by reducing a non-incremental 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 non-incrementality relative to the class of locally persistent algorithms, we must show that the transformation algorithms referred to in (2) are locally persistent.

Proposition 4.5. *Let* $\mathbf{R} = (S, \oplus, \otimes, \bar{0}, \bar{1})$ *be an arbitrary closed semiring. The SS-***R** *problem is non-incremental 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 $(G, s, \lambda e.\bar{1})$. In the target problem instance, the summary value at v, $C(s,v)$ is $\bar{1}$ if v is reachable from s, and $\bar{0}$ otherwise.

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

It follows from the above proposition that SSSP≥0 is a non-incremental 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 non-incremental. For example, graphs with 0-weight cycles are what causes SSSP≥0 to be non-incremental. 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 non-incremental. 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:

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

A particular data-flow analysis problem P is specified by a particular semilattice L and a constant c (which is often a greatest or least element of the semilattice). 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 non-incremental by reducing SS-REACHABILITY to P.

Proposition 4.6. *All meet-semilattice data-flow problems are non-incremental for locally persistent algorithms.*

Proof. Consider any meet-semilattice data-flow problem P with associated semilattice L and constant c. 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 non-incrementality of SS-REACHABILITY that P is non-incremental. □

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:

$S(s) = c$
$S(u) = \top$ for $u \neq s$.

Hence, the edge-labeling M from the reduction used in the proof of Proposition 4.6 is, in fact, realistic.

In conclusion, note that the reduction used in the proof of Proposition 4.6 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 incremental single-source reachability problem. In other words, for a class of algorithms to have members that are incremental for any data-flow analysis problem, there must be an algorithm of the class that solves the single-source reachability problem incrementally.

5. A COMPUTATIONAL-COMPLEXITY HIERARCHY FOR INCREMENTAL 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 incremental-computation problems are classified according to their incremental complexity with respect to locally persistent algorithms. The computational-complexity hierarchy for incremental problems is depicted in Figure 10. In the remainder of this section we describe the results from other papers that have a bearing on this way of classifying incremental problems; some additional discussion of these problems can be found in Section 6.

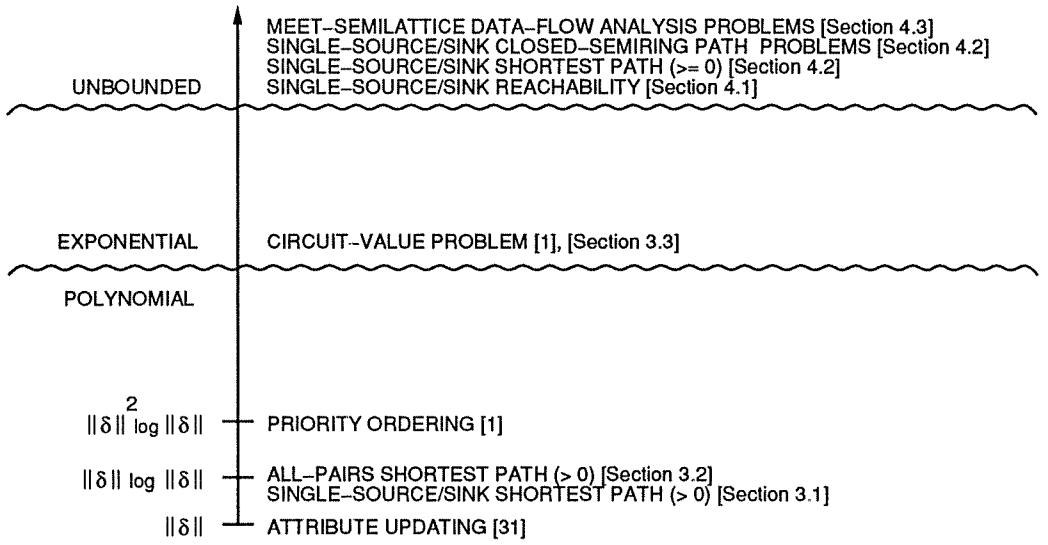


Figure 10. The computational-complexity hierarchy for incremental-computation problems that exists when problems are classified according to their incremental complexity in terms of the parameter $\|\delta\|$ with respect to locally persistent algorithms. (In the figure, we do not distinguish between $\|\delta\|_1$ and $\|\delta\|_2$. $\|\delta\|$ represents $\|\delta\|_1$ in all cases except for APSP>0.)

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 $\|\delta\|$ [30] (see also [31] and [32]). Thus, the algorithm he gave for the problem is asymptotically optimal. ([30] 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 incremental circuit-value problem and the problem of maintaining a priority ordering in a DAG. In the incremental 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 $priority(v) > priority(w)$.[8] 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 $O(\|\delta\|^2 \log \|\delta\|)$.

(2) They showed that any locally persistent incremental algorithm for the incremental circuit-value 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 *Exp*-time incremental problems from the class of *P*-time incremental problems. Recall that in Section 3.3 we give two bounded algorithms for the incremental circuit-value problem (where the bound is an exponential function of $\|\delta\|$). Previous to our work, no bounded algorithm for the incremental circuit-value 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 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 incremental-computation 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 incremental update algorithms"[39]; similar remarks have appeared in several other papers. However, our work shows that for some incremental-computation 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 $O(\|\delta\| \log \|\delta\|)$.

---

[8]Note that the incremental 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.

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 incremental problems has been addressed in two previous papers, one by Reif [29] 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 [29].[9] At the end of the paper, Reif lists a number of incremental 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 incremental *decision* problems, the reductions that we present in Section 4 are among incremental *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 is 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 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 incremental decision problems in terms of $\|\delta\|$.

A different approach to the problem of classifying incremental problems was proposed in a paper by Berman, Paull, and Ryder [3]. Berman, Paull, and Ryder classify incremental problems through the notion of an *incremental relative lower bound* (IRLB). An IRLB relates the worst-case time required for an incre-

---

[9]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.

mental 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 incremental-computation 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 [30-32]. 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" incremental problem).[10] 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).

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

[10]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 [14] to establish that no incremental algorithm for the all-pairs single-source shortest-path problem can do better in the worst case than the best batch algorithm for the problem.

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 $v \rightarrow sink(G)$. 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 $O(\|\delta\| \log \|\delta\|)$.

The fact that SSSP>0 has an $O(1)$ IRLB *and* a 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 incremental 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 incremental-computation 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. It is true that 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. However, 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 [30] (see also [31] and [32]).[11] 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 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 [32] and ordered attribute grammars [36,46].

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.

[11] In these papers, the parameter $\|\delta\|$ is referred to as |AFFECTED|.

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 $O((\|\delta\| + k) \cdot \log n)$ [33], 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 incremental 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 $O(\|\delta\|^2 \log \|\delta\|)$.

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

(3) They gave an (unbounded) algorithm for the incremental circuit-value problem that used their incremental 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 incremental circuit-value problem is unbounded.

The results that we present in this paper can be compared to those of Alpern *et al.* as follows:

(1) We show that three additional problems[12]—SSSP>0, APSP>0, and the circuit-value problem—have bounded incremental algorithms. SSSP>0 and APSP>0 are shown to be P-time incremental; the circuit-value problem is shown to be *Exp*-time incremental. Previous to our work, no bounded algorithm for the incremental circuit-value problem was known. (It should be noted that our circuit-value algorithm does not address the problem of determining whether an edge insertion introduces any cycles.)

(2) We establish a different kind of lower bound from that established by Alpern *et al.* By looking at graph problems in which the graphs may have cycles, we show that more difficult problems than the incremental circuit-value problem exist (from the standpoint of incremental computation): for these problems, *no* bounded locally persistent incremental algorithm exists.

---

[12]Actually, using ideas very similar to those in our algorithm for SSSP>0, we also developed a bounded incremental algorithm for the problem of maintaining the longest path to a distinguished sink vertex in a DAG. Again, the basic idea is to maintain a distinguished subset of the edges in G's edge set, denoted by $LP(G)$; $LP(G)$ contains exactly those edges that occur in some longest path from some vertex of G to $sink(G)$. In other words, the graph $(V(G), LP(G))$ is the graph of longest paths to $sink(G)$. Because this algorithm is similar in spirit to the algorithm we give for SSSP>0, and because of length considerations, we refrained from including the algorithm for maintaining longest paths in this paper.

6.3. Previous Work on Incremental Shortest-Path Algorithms

Section 3.2 of this paper presents a bounded incremental algorithm for the all-pairs shortest-path problem with positive edge weights (APSP>0) (assuming the collection of vertices is fixed in advance). There have been three previous papers on APSP>0—by Dionne [11], Rohnert [37], and Even and Gazit [14]—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 incremental *problem*, with quantities that reflect the updating costs for various internal data structures that store auxiliary information used by a particular *algorithm* for the incremental 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 incremental problem itself; one must ask how these costs compare with $\|\delta\|$.) For example, with both Rohnert's and Even and Gazit's algorithms, 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.*

An updating method that is similar in spirit to our APSP>0 algorithm from Section 3.2 was proposed by Dionne in 1978 [11]. Dionne studies both the symmetric and asymmetric cases of the all-pairs problem. (In the symmetric case, $length(i, j) = length(j, i)$ for all vertices i and j.) Dionne cites Murchland as the source of the algorithm he studies for the asymmetric case [25] and Boyce, Farhi, and Weischedel as originators of a similar algorithm to his for the symmetric case [4].

Dionne observed that to update distances after the insertion or deletion of edge $v \longrightarrow w$, it is useful to compute two sets N_1 and N_2 such that $N_1 \times N_2$ includes all ordered pairs (a, b) such that the length of the shortest path from a to b changes [11]. These sets are akin to the sets referred to as AffectedSources and AffectedSinks, respectively, in Figures 4 and 6. However, because Dionne did not have a bounded incremental algorithm for SSSP>0 at his disposal, he did not have a bounded method for determining AffectedSources and AffectedSinks, and as a result the algorithm he gave for APSP>0 is not bounded.

In the case of edge insertion, although Dionne's method is able to identify the sets AffectedSources and AffectedSinks, the algorithm he gives to actually find these sets uses loops of the form:

> **for** each vertex k of the graph's vertex set **do** . . . **od**

(see pages 136 and 143 of [11]), and hence his algorithm has complexity $\Omega(|V(G)|)$. Consequently, the algorithm's complexity is not bounded by any function of $\|\delta\|_i$ for any fixed value of i.

In the case of edge deletion, Dionne's method *overestimates* AffectedSources and AffectedSinks. Furthermore, the algorithm he gives to find these sets again uses loops of the form:

> **for** each vertex k of the graph's vertex set **do** . . . **od**

(see pages 139 and 145 of [11]), and hence his edge-deletion method also has complexity $\Omega(|V(G)|)$.

In contrast, the procedures $\text{DeleteEdge}_{APSP>0}$ and $\text{InsertEdge}_{APSP>0}$ that we give in Section 3.2 can determine AffectedSources and AffectedSinks *exactly*. As a consequence, the time complexity of procedure $\text{DeleteEdge}_{APSP>0}$ is $O(\|\delta\|_2 + \|\delta\|_1 \log \|\delta\|_1)$, and the time complexity of procedure $\text{InsertEdge}_{APSP>0}$ is $O(\|\delta\|_1)$. Thus, our results show that APSP>0 is a P-time incremental problem.

Although the algorithms Dionne studies are unbounded, he reports that they have excellent performance. His experiments indicate speedups of between 40 and 250 for an edge-length reduction and between 7 and

20 for an edge-length increase [11].[13] In all cases, the speedup reported is with respect to the time required to recompute all distances from scratch using Floyd's shortest-path algorithm [15].

Lin and Chang have given a bounded incremental algorithm for APSP>0 when the problem is restricted to the case of *edge insertions only* [22]. The algorithm they give uses an auxiliary procedure that is very similar to InsertUpdate; they employ a different method for determining the sinks for which calls must be made on InsertUpdate. The total cost of their edge-insertion procedure is linear in $\| \delta \|_1$. The extension of their method to handle edge deletions was left as an open problem.

In addition to maintaining the distance matrix for the graph, the algorithms of Rohnert, Even and Gazit, and Lin and Chang 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 DeleteEdge$_{APSP>0}$ and InsertEdge$_{APSP>0}$ 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 Halder [17], Pape [26], Spira and Pan [43], Cheston [8], Cheston and Corneil [9], and Ausiello *et al.* [2]; 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, none of the algorithms presented in these papers is bounded. The only bounded incremental algorithm that has been given previously is the algorithm of Lin and Chang for the special case of APSP>0 restricted to edge insertions only [22].

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 [5,6] and by Gurevitch and Shelah [16]. In this paper, we measure the complexity of an *incremental* algorithm in terms of the sum of the sizes of the *changes* in the input and the output.

There has been a variety of work carried out on the problem of taking a high-level specification of a computation to be performed and creating an incremental evaluator that maintains the result value of the computation as inputs to the computation are varied [20,21,27,42,44,47]. From the standpoint of computational complexity, the most interesting result from this group of papers is the technique that Pugh developed for supporting incremental computation using function caching [28] (see also [27]). Pugh's work is based on the notion of a *stable decomposition* of a data structure; for example, he developed a clever representation of sets in which each set is—with high probability—represented by a balanced tree. The decomposition can be exploited by divide-and-conquer algorithms that map an associative function over the elements of the set. The decomposition has the property that when a constant number of elements in set S are changed, the decomposition of the new set S' has, with high probability, only $O(\log |S'|)$ subproblems that did not arise in the decomposition of S. Thus, by caching the previously computed results of the subproblems on S, only $O(\log |S'|)$ subproblems need be solved to update the computation of the function on S to reflect the value of the function applied to S'. Note that Pugh's method is "unbounded" in the sense of the term used in this paper: the update time is a function of the size of the input—$O(\log |S'|)$—rather than a function of the sum of the sizes of the changes in the input and the output.

A number of papers in the literature on incremental algorithms concern incremental data-flow analysis [7,23,24,38,39,48]. 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

[13]Recall that with our algorithms, edge-length reductions can be handled by generalizing InsertEdge$_{APSP>0}$ slightly, while edge-length increases can be handled by generalizing DeleteEdge$_{APSP>0}$ slightly.

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 an incremental 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 incremental 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 incremental 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 incremental 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 incremental algorithms.

As a final closing remark, it should be noted that although most incremental 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 [7] performs well in practice [40]; as discussed earlier, Dionne reports excellent performance for some unbounded algorithms for APSP>0 [11].

ACKNOWLEDGEMENTS

We are grateful for the comments and helpful suggestions of Susan Horwitz, Alan Demers, and Tim Teitelbaum.

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. Boyce, D.E., Farhi, A., and Weischedel, R., "Optimal network problem: a branch-and-bound algorithm," *Environment and Planning*, (5) pp. 519-533 (1973). As cited in reference [11].
5. 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).
6. 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).
7. 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).
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. Dijkstra, E.W., "A note on two problems in connexion with graphs," *Numerische Mathematik* **1** pp. 269-271 (1959).
11. Dionne, R., "Etude et extension d'un algorithme de Murchland," *INFOR* **16**(2) pp. 132-146 (June 1978).
12. 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].
13. Even, S. and Shiloach, Y., "An on-line edge-deletion problem," *J. ACM* **28**(1) pp. 1-4 (January 1981).
14. Even, S. and Gazit, H., "Updating distances in dynamic graphs," *Methods of Operations Research* **49** pp. 371-387 (1985).
15. Floyd, R.W., "Algorithm 97: shortest path," *Commun. of the ACM* **5** p. 345 (1962).
16. Gurevich, Y. and Shelah, S., "Time polynomial in input or output," *J. Symbolic Logic* **54**(3) pp. 1083-1088 (September 1989).
17. Halder, A.K., "The method of competing links," *Transportation Science* **4** pp. 36-51 (1970).
18. Henderson, P. and Weiser, M., "Continuous execution: The Visiprog environment," in *Proceedings of the Eighth International Conference on Software Engineering*, (1985).
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. Horwitz, S. and Teitelbaum, T., "Generating editing environments based on relations and attributes," *ACM Trans. Program. Lang. Syst.* **8**(4) pp. 577-608 (October 1986).
21. Koenig, S. and Paige, R., "A transformational framework for the automatic control of derived data," pp. 306-318 in *Proceedings of the Seventh International Conference on Very Large Data Bases*, (Cannes, France, September 1981), (1981).
22. Lin, C.-C. and Chang, R.-C., "On the dynamic shortest path problem," *Journal of Information Processing* **13**(4)(1990).
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., "A fixed matrix method for all shortest distances in a directed graph and for the inverse problem," Doctoral dissertation, Universität Karlsruhe, Karlsruhe, Germany (). As cited in reference [11].
26. Pape, U., "Netzwerk-veraenderungen und korrektur kuerzester weglaengen von einer wurzelmenge zu allen anderen knoten," *Computing* **12** pp. 357-362 (1974).
27. Pugh, W. and Teitelbaum, T., "Incremental computation via function caching," pp. 315-328 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages*, (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).
28. Pugh, W.W., "Incremental computation and the incremental evaluation of functional programs," Ph.D. dissertation and Tech. Rep. 88-936, Dept. of Computer Science, Cornell University, Ithaca, NY (August 1988).
29. Reif, J.H., "A topological approach to dynamic graph connectivity," *Information Processing Letters* **25**(1) pp. 65-70 (1987).
30. 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).
31. 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).
32. Reps, T., *Generating Language-Based Environments*, The M.I.T. Press, Cambridge, MA (1984).
33. 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).

34. Reps, T. and Teitelbaum, T., "Language processing in program editors," *Computer* **20**(11) pp. 29-40 (November 1987).
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. 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. Melhorn,Springer-Verlag, New York, NY (1985).
38. 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).
39. Ryder, B.G. and Paull, M.C., "Incremental data flow analysis algorithms," *ACM Trans. Program. Lang. Syst.* **10**(1) pp. 1-50 (January 1988).
40. 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).
41. Sedgewick, R., *Algorithms*, Addison-Wesley, Reading, MA (1983).
42. Shmueli, O. and Itai, A., "Maintenance of views," pp. 240-255 in *Proceedings of ACM SIGMOD 84*, (Boston, MA, 1984), (1984).
43. Spira, P.M. and Pan, A., "On finding and updating spanning trees and shortest paths," *SIAM J. Computing* **4**(3) pp. 375-362 (September 1975).
44. Sundaresh, R.S. and Hudak, P., "Incremental computation via partial evaluation," in *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages*, (Orlando, FL, January 1991), ACM, New York, NY (1991).
45. Tarjan, R.E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA (1983).
46. Yeh, D., "On incremental evaluation of ordered attributed grammars," *BIT* **23** pp. 308-320 (1983).
47. Yellin, D. and Strom, R., "INC: A language for incremental computations," *ACM Trans. Program. Lang. Syst.* **13**(2) pp. 211-236 (April 1991).
48. 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).