LOWER AND UPPER BOUNDS FOR INCREMENTAL ALGORITHMS

BY ARTHUR MICHAEL BERMAN

A dissertation submitted to the
Graduate School—New Brunswick
Rutgers, The State University of New Jersey
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
Graduate Program in Computer Science

Written under the direction of
Marvin C. Paull
and approved by

_____

_____

_____

_____

_____

New Brunswick, New Jersey

October, 1992

ABSTRACT OF THE DISSERTATION

Lower and Upper Bounds for Incremental Algorithms

by Arthur Michael Berman, Ph.D.

Dissertation Director: Marvin C. Paull

An incremental algorithm (also called a dynamic update algorithm) updates the answer to some problem after an incremental change is made in the input. We examine methods for bounding the performance of such algorithms. First, quite general but relatively weak bounds are considered, along with a careful examination of the conditions under which they hold. Next, a more powerful proof method, the Incremental Relative Lower Bound is presented, along with its application to a number of important problems. We then examine an alternative approach, δ-analysis, which had been proposed previously, apply it to several new problems and show how it can be extended. For the specific problem of updating the transitive closure of an acyclic digraph, we present the first known incremental algorithm that is efficient in the δ-analysis sense. Finally, we critique the existing approaches to incremental lower bounds, examining their strengths and weaknesses in light of the general goal of lower bounding incremental algorithms, and considering under what conditions which method(s) of analysis are most appropriate and useful.

Acknowledgements

One can survive 28 years of education only with a lot of help.

First, I am indebted for the love and guidance of my parents Arthur Berman and Elaine Berman. Their interest in the world and dedication to learning gave me the foundation for all my academic (and other) endeavors.

I have been extremely lucky to have had many wonderful teachers. Some that come to mind from my days in Los Angeles public schools are Mrs. Hartman, Mrs. Avins, Mr. Wendt (who first taught me programming in 1969), Mr. Farnsworth, Mr. Chasman, Mr. Farnham, Mr. Shoenberg, and Mr. Layton. At Pomona College I was fortunate to learn from James Likens, Monica Morris, and Sandy Grabiner. One of my best experiences was studying all-too-briefly with Margaret Paul at Lady Margaret Hall, Oxford, England.

I came to Rutgers University through a sequence of events that had little to do with any reasonable expectation that it would be a good place to study Computer Science. I have had an extraordinarily high percentage of outstanding and committed teachers. More importantly, I was treated as a colleague and member of the profession of Computer Science long before I had done anything to earn such treatment. I guess they thought that if they treated me like I already was a Computer Scientist, I would have no choice but to become one. In addition to the members of my committee, who are thanked below, I would like to mention: Matthew Morgenstern, who first introduced me to data structures; Ann Yasuhara, who took me under her wing and taught me that research is a community activity; Ken Kaplan, who taught me a great deal about teaching and dedication, and always gave me respect even when I may not have deserved it; and Bill Steiger, who was generous with his time and his encouragement even when I wasn't prepared to appreciate it. I am also thankful for the graduate fellowship provided

by Rutgers University for part of my tenure, and for NSF support of my work with Ann Yasuhara. Other Rutgers faculty I benefited from studying with include Yehoshua Perl, Diane Souvaine, Abe Lockman, Joan Lucas, Mike Saks, and Endre Szemeredi.

I gratefully acknowledge the support and assistance of graduate secretaries Janet Willis and Valentine Rolfe (and thanks for the Pecan Sandies!)

Some students are able to complete dissertations under the direction of advisors with whom they have no personal relationship. I don't believe that this would have been possible for me. Marv Paull has been my advisor, my mentor, and my friend. His door was always open, even when the distractions of his own life would have made it easy for him to hide from me. We've been working together for ten years, and I'm glad that we succeeded in the end.

I might never have finished this work but for the tremendous help and encouragement of Barbara Ryder. Having her on my committee was like having two advisors, and believe me, I needed both of them. She is one of the most giving people I've known inside or outside academe. Tom Marlowe was extremely helpful in the writing of this dissertation. From correcting many errors of fact to helping me think about some of the grander issues and implications of my work, Tom's energy, enthusiasm, and wit never failed. Don Smith was a good teacher, essential support, and a great guy to talk to. Finally, the last minute entry of Lori Pollock was the final stroke of luck in the process. Her generosity and commitment were much appreciated.

There are the many friends, family members, and other supporters who have kept me sane through my long years of graduate study: Tom and Alice Walker, Mark Biggers, Thom Wolke, Bob Yahn, Clearwater, John Bresina, Patricia Riddle & Mike Barley, Fritz Henglein, Carlos Yabut, Michael Platoff, Tom Ostrand, Jody Gevins, Jim Llewellyn, Martin Carroll, Suzanne Menzel Orr, Robert Corn, Smadar Kedar, Jeffrey Aaron & the New Brunswick Religious Society of Friends, William & Susan Cohen, Mary Laude, Built for Comfort, Lori Pratt & John Smith, Emmi Schatz, Yong-Fong Lee, Steve Masticola, Hemant Pande, Phil Franklin, Bob Barker, D.U.C.K., Jeff Salowe, Jeannine Fay.

I also received a lot of support from Glassboro State College, including release

time from some of my teaching responsibilities. I thank my Deans (Minna Doskow, Pearl Bartelt), members of my Department (Seth Bergmann, Jianning Xu, Khaled Amer, Evelyn Amato) and Academic Computing (Jack Cimprich, Debbie Pillow, Leigh Weiss), and my many enthusiastic and forgiving students, in particular Bruce Klein and Genevieve Millsap.

I thank my son Adam for lots of two hour naps during which I wrote most of the text of this dissertation; and I thank those that provided childcare, in some cases on short notice, so that I'd have time to write and to make trips to Rutgers and Delaware: Jeanine Mason, Jeannine Fay, Alice Walker, Genevieve Millsap.

Finally, I thank my patient and ever-loving wife Ann Walker, who made many compromises so that I could complete this multi-decade project, and who always believed I would succeed.

Dedication

For my partner Ann, and our incremental addition Adam.

Table of Contents

List of Figures

Chapter 1

Introduction

In recent years there has been a great deal of interest in what are known as incremental, dynamic, or on-line algorithms. The idea is to develop algorithms that can adjust their answers efficiently in response to changes in the input data. Domains for such algorithms have included:

- graph theoretic algorithms:
  - connectivity [ES81, Har83, Che84],
  - spanning trees [SP73, CH78, FS84, Fre85],
  - spanning forests [Wes89],
  - shortest paths [Rod68, Che76, GSV78, Fuj81, CC82, Gaz83, EG85, AMSN89, AIMSN90, Ita91],
  - biconnected components [Sac86, WT92, BT90],
  - triconnected components [Ita91, BT90],
  - transitive closure [IK83, Ita86, Ita88, LPv88, YS88, Yel91],
  - planar graphs [Tam88, TP90, BT89, EIT+92, PT88];
- computational geometry [Ov81, CBT+92];
- data bases [ABJ89];
- syntax-directed editors and grammars [Rep82, RTD83, Rep88, ACR+87];
- data-flow analysis [Ryd83, RP88, RC86, CR88, Mar89, Bur90, PS89, Zad84, Gho83, KRvM88],
- code generation and optimization [Pol86],

- polynomial satisfiability problems [AI91].

There have also been parallel incremental algorithms developed for minimum spanning trees and connected components [PR85].

In comparison to the extensive literature on incremental algorithms, there is a relative paucity of work in the area of lower bounds. In their paper describing an incremental algorithm for updating minimum paths, Even and Gazit prove that in the worst case no incremental algorithm that handles edge deletions can be faster than an algorithm that solves the problem from scratch [EG85]. This work inspired the developments described in Chapter 4. Work begun by Alpern et al. [AHR+90] and extended by Reps and Ramalingam [RR91] gives lower bounds based on δ-analysis, within a limited model of computation called local persistence; their work is described and extended further in Chapter 5.

The contributions of this dissertation are as follows:

- to define a framework in which incremental lower bounds can be derived;
- to present a new general technique, Incremental Relative Lower Bounds (IRLB's), which can be used to derive lower bounds for many of the important incremental problems in the literature;
- to extend the applicability of δ-analysis;
- to analyze the Transitive Closure problem via IRLB and δ-analysis, and to present a new algorithm for updating Transitive Closure on a Directed Acyclic Graph;
- and to compare and contrast the applicability, strengths and weaknesses of IRLB's and δ-analysis.

The dissertation is comprised of the following chapters:

1. *Introduction:* this introduction.
2. *Definitions:* some fundamental definitions for incremental algorithms.

3. *A General Lower Bounds Result:* Relatively weak but general bounds that apply to nearly all incremental algorithms are derived; some of the material in this chapter appeared previously in [PaCB84].

4. *Incremental Relative Lower Bounds:* The method of Incremental Relative Lower Bounds (IRLB's) is presented and applied to many incremental problems; the material in this chapter appeared previously in [BPR90].

5. *δ-Analysis of Incremental Algorithms:* The δ-analysis approach to analyzing incremental algorithms is presented; new results in this model are derived.

6. *Transitive Closure — A Case Study in Incremental Bounds:* Lower bounds for the problems of transitive closure on a directed graph and transitive closure on a directed acyclic graph (DAG) are derived using IRLB analysis, and the first known δ-bounded, fully dynamic, incremental algorithm for transitive closure on a DAG is presented.

7. *Comparing Different Models — What Do We* Really *Want To Know:* The available methods for lower-bounding incremental algorithms are compared and contrasted relative to the desired goals.

8. *Conclusion:* The results of this dissertation are summarized. Open problems and future directions are assessed.

Chapter 2

Definitions

2.1 Terms and Definitions for Incremental Algorithms

Since the study of incremental algorithms is relatively new, and has been undertaken by researchers in several computer science specialties (primarily, but not exclusively, algorithms and programming languages), there is no standard vocabulary for this area. For example, what we refer to here as "incremental algorithms" have also been called "dynamic algorithms," "on-line update algorithms," and "on-line maintenance algorithms." Not only do the names change, but also the exact definition of what constitutes an incremental change; in particular, some algorithms are analyzed for changes in which a single run of the algorithm can make numerous updates, while others allow no more than an "atomic" change. While we cannot hope that this dissertation will clean up this semantic mess, this chapter carefully defines the exact meaning of "incremental algorithm" within this dissertation. These definitions are based on those in [PaCB84] and [BPR90].

Definition 2.1.1 Let α:$P \rightarrow Q$ be a function with domain P being the set of *problem instances* or *inputs*, and range Q the set of *answers* or *outputs*. Each $p \in P$ and $q \in Q$ is itself a set, with $|p|$, the length of the problem instance, represented by ℓ.[1] Let A be an *algorithm* that, given a problem instance p as input, returns q such that $q = \alpha(p)$; we say A *implements* α. The number of steps required by algorithm A to compute $\alpha(p)$, in the worst case, is the *(time) complexity* of A, denoted $T_A(\ell)$. When we measure the

[1] In most presentations the problem length is assumed to be n, but this causes confusion when discussing graphs, where we use n to represent the number of vertices.

complexity of our algorithms, we do *not* count the time required to input the problem.[2] T_α represents the time required by the (time) optimal algorithm for α.

Choose α, p, and q. Let p' be another problem instance such that p' differs from p by exactly one input element. This difference can represent an addition, a deletion, or a change (a deletion plus an addition). Let the change in p be represented by Δp. If given p, q, and Δp — that is, the old input set, the old solution, and the change — algorithm ΔA determines q' such that $q' = \alpha(p')$, then ΔA is called an *incremental algorithm* for function α. We allow ΔA to return either q' in its entirety, or to return the difference between q and q'.

When applying Definition 2.1.1 to a function α, we note that the length of the problem is $|p|$. For example, in a typical graph problem it is consistent with our model to treat pairs of vertices as the elements in the input set; in this model, considering each vertex as an individual element would not be allowed, since a single vertex could be an element in $n-1$ vertex pairs. Hence, the removal or addition of a vertex in a graph would not be regarded as a single incremental change, but rather as a sequence of incremental changes involving each edge incident on the vertex.

2.2 Graphs

The following standard definitions will be used in various places in the dissertation. For further information on graphs, see an algorithms or data structures reference, e.g., [Tar83, CLR91].

Definition 2.2.1 Let $G = (V, E)$ be a graph, where V is a set representing the vertices, and E is the set of edges. Each edge is a set of size two $\{u, v\} \in E$, where $u, v \in V$ and $u \neq v$. Generally, we write uv to represent the edge connecting u to v. The size of the graph is $n + m$, where $n = |V|$ and $m = |E|$.

A directed graph (digraph) is defined as follows: Let $G = (V, E)$ be a digraph, where V is a set of vertices, E a set of edges, and each edge (u, v) is an ordered pair. We say

[2] This is a technicality, but its importance will be seen later. Implicitly, we are assuming that the ℓ inputs are available in random access memory when the counting begins.

that (u, v) is directed from u to v. Again, we write uv to represent edge (u, v) in the graph.

A *path* from u to v in a graph or a digraph is a sequence $(v_0, v_1, \ldots, v_k)$, where $v_i v_{i+1} \in E$ for $0 \leq i < k$, $v_0 = u$ and $v_k = v$. If there is a path from u to v, we write $u \rightsquigarrow v$.

2.3 Asymptotic Growth Notation

We use the standard definitions as given in the algorithms textbook of Cormen, Leiserson, and Rivest [CLR91].

For all the following definitions, c, c_1, c_2, and n_0 represent positive constants, and $g(n)$ any given function.

Definition 2.3.1 Define $\Theta(g(n))$ to be the set of functions:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$$

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Definition 2.3.2 Define $O(g(n))$ as the set of functions:

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}$$

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

Definition 2.3.3 Define $\Omega(g(n))$ to be the set of functions:

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}$$

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

Note that for any two functions $f(n)$ and $g(n)$, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Definition 2.3.4 To denote an upper bound that is not asymptotically tight, we use $o(g(n))$, the set

$$o(g(n)) = \{f(n) \mid \text{ for any } c > 0, \exists n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n), \forall n \geq n_0\}$$

Definition 2.3.5 A lower bound that is not asymptotically tight is denoted by the set $\omega(g(n))$:

$$\omega(g(n)) = \{f(n) \mid \text{for any } c > 0, \exists n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n), \forall n \geq n_0\}$$

Chapter 3

A General Lower Bounds Result

3.1 Preliminaries

Given some rather plausible restrictions, it is possible to show that no incremental algorithm for a problem P can be more than ℓ times faster than the best algorithm for solving P from scratch. This is hardly surprising; what is surprising is that despite the fact that this result is widely assumed (see, e.g., [Che76]), the conditions under which this result does and does not hold true have not been carefully considered.[1]

Typically, a good incremental algorithm maintains some auxiliary information about the operation of the algorithm from one incremental change to the next. In general, an incremental algorithm returns the tuple (q, h), where q is the current answer and h is any auxiliary information. We refer to h as the *history* of the algorithm. The most complete set of information about the operation of an algorithm is the sequence of state vectors encountered in the computation of the result; call this the *complete history*, $\overline{h}$. The computation model used determines the form of the history. For a finite state machine, $\overline{h}$ is the sequence of states encountered. For a Turing machine, it is the contents of the tape(s) and the internal state of the machine after each step. In a Random Access Machine, $\overline{h}$ consists of the complete contents of memory and registers after each step. Note that q depends only upon the definition of the problem being solved, while h depends both upon the computation model and the particular algorithm being used. Note also that the requirement of producing a history as output does not necessarily increase the cost by more than a constant factor, since the algorithm must already maintain its state internally. However, this may not be the case if the

[1] Some of the material in this chapter appeared in [PaCB84].

entire state is "dumped" at each step; instead, a record of changes in state should be written as output, from which the sequence of states can be constructed later. Furthermore, certain incremental algorithms may require that the history be organized in a particular form — a data structure — which may add to the time required by the initial algorithm. The additional cost of establishing supplementary data structures used by the incremental algorithm, above and beyond that required to produce the initial solution, is often referred to as preprocessing cost. This preprocessing cost can be viewed as part of the cost of producing h.

Lemma 3.1.1 *Given any algorithm A, problem instance p, answer q, and history h, the following must hold:*

$$T_A(\ell) = \Omega(|q| + |h|).$$

Proof In order to "return" q and h, A must write them to some memory device. Clearly, the time required to do this is proportional to the number of elements in q and h. ■

3.2 A Lower Bounds Theorem

We consider the question of bounds on the speed of an incremental algorithm. If there exists a fast incremental algorithm for a problem, it ought to be possible to build a fast non-incremental algorithm by using a series of calls to the incremental algorithm. This procedure would be organized as follows:

1. Use some initial dummy input value, and compute the output value for this dummy input value.

2. Change the initial values, one by one, to the real values for the problem instance to be computed, and apply the incremental algorithm each time.

This is expressed more formally as procedure A' in Figure 3.1.

0. $p \leftarrow$ *initial dummy input of length* ℓ
1. $q \leftarrow \alpha(p)$
2. $h \leftarrow$ *initial history for input* p
3. **for** $i \leftarrow 1$ **to** ℓ **do**
4. $\quad \{q, h\} \leftarrow \Delta A(q, p, h, p_i)$
5. $\quad p \leftarrow p$ updated by p_i

Figure 3.1: Procedure A'

Steps 3 and 4 of procedure A' clearly require time $\ell \times$ complexity of step 4, $\leq \ell T_{\Delta A}(\ell)$. Assuming that steps 0, 1, and 2 are fast relative to T_α, it is not possible for ΔA to be of complexity $< T_\alpha/\ell$ because otherwise procedure A' would be a procedure for computing α which is faster than the fastest (non-incremental) algorithm, a contradiction. It now remains to examine under what conditions steps 0, 1, and 2 will be relatively fast, because in such cases we can conclude that the complexity of every such incremental algorithm for α is bounded from below by T_α/ℓ.

Definition 3.2.1 Consider a function α, with the property that for any ℓ, there exists a problem instance p_0^ℓ, such that the instance can be constructed and $\alpha(p_0^\ell)$ determined in time dominated by T_α. A function for which such a procedure exists is said to have a *fast initialization value.*

A fast initialization value can be found by inspection for many functions; typically, the initialization time required is $O(\ell + \text{size of output})$. For example, if the problem is to sort n elements then a fast initialization value is the tuple of n elements $(1, \ldots, n)$, which is already sorted. Similarly, a minimum or maximum value can be found in such a list by setting up the ordered list and choosing the first or last value. Thus sorting, maximum, and minimum have fast initialization values. Many problems involving weighted graphs, such as spanning tree and maximum flow, have immediate and arbitrary solutions when all weights are set to 0 or to ∞, depending upon the problem. For example, if the problem is to find a minimal spanning tree, a fast initialization value is a complete graph on n nodes with all edges of weight zero; a minimum spanning tree of this graph is $\{v_1v_2, v_2v_3, \ldots, v_{n-1}v_n\}$. Note that the existence of a fast initialization is a property

of the function, and not of any particular algorithm for its implementation. Also note that since the notion of history has to do only with algorithms, there is no direct relationship between history and fast initialization values.

Definition 3.2.2 If an algorithm A and associated incremental algorithm ΔA implement α using a history that is never more expensive to produce than the output q, then A, and ΔA are algorithms with *restricted history*.

Algorithms that produce no history are a special case of those that work with a restricted history. Lemma 3.1.1 bounds the size of such a history, i.e., $|h| \leq T_A$. For the purposes of the theorem proved below, it would suffice to combine the above two definitions into one, and just consider algorithms where all the initialization can be done quickly. However, the two restrictions are fundamentally different, since the existence or non-existence of a fast initialization is implicit in the definition of a function, while the time to produce a history depends upon the particular implementation of that function.

We can now prove a theorem showing that when a function has a fast initialization value, we can bound the speed of all incremental algorithms with restricted history, for that function.

Theorem 3.2.3 *Consider some function α and those algorithms (non-incremental and incremental) with restricted history. If α has a fast initialization, and if ΔA is an incremental algorithm for α, then:*

$$T_{\Delta A}(\ell) \geq \frac{T_\alpha}{\ell}.$$

Proof Consider procedure A' for computing $\alpha(p_\ell)$, $|p| = \ell$, as shown in Figure 3.1. Now consider $T_{A'}(\ell)$, the time complexity of A'. By the fast initialization assumption, the complexity of steps 0 and 1 is $< T_\alpha$; by the restricted history assumption, the complexity of step 2 can be no more than that of step 1. The complexity of steps 3 and 4 is $\ell \times$ complexity of step $4 \leq \ell T_{\Delta A}$.

Conversely, assume that the theorem does not hold; that is,

$$T_{\Delta A}(\ell) < \frac{T_\alpha}{\ell}.$$

Then the complexity of algorithm A' is $< T_\alpha$; but this is a contradiction, hence the theorem holds. ■

To illustrate the above proof, consider the problem "sort by comparisons." The tuple $(1, \ldots, n)$, which is sorted, is a fast initialization value. Suppose there is an incremental algorithm for the problem with complexity less than $\log \ell$ per update. Then, apply this incremental algorithm ℓ times, adding the "real" elements that need to be sorted one at a time. After ℓ steps, the list will be sorted, in total time $o(\ell \log \ell)$. But this is clearly a contradiction since comparison sorting *cannot* be that fast (see, e.g., [HS76, pp. 350–352]). So the best incremental comparison sorting algorithm (with restricted history) cannot be faster than $\log \ell$.

3.3 Bounding Amortized Complexity

In amortized complexity, the time per operation is averaged over a worst-case sequence of operations [Tar84]. The advantage of amortized analysis is that in some cases, the average time per operation in the worst-case sequence is less than the worst-case time. Theorem 3.2.3 can be viewed as applying to the amortized case as well as the worst-case. We will use the notation $T^n_{\Delta A}$ to represent the time per operation required by an incremental algorithm ΔA, amortized over n operations.

Corollary 3.3.1 *Consider a function α and algorithms with restricted history. α has a fast initialization, and if ΔA is an incremental algorithm for α, then*

$$T^\ell_{\Delta A}(\ell) \geq \frac{T_\alpha}{\ell}.$$

Proof If, to the contrary, $T^\ell_{\Delta A}(\ell) < T_\alpha/\ell$, then ΔA can be applied ℓ times at a cost $< T_\alpha$; by the same argument as in Theorem 3.2.3, this can be seen to be impossible; hence the theorem must hold. ■

3.4 *NP*-complete Incremental Algorithms

Now we apply Theorem 3.2.3 to an *NP*-complete problem, 3-SAT, and bound the speed of incremental algorithms for this problem. 3-SAT is the problem of determining whether a set of clauses, each clause containing exactly three literals, has a satisfying truth assignment [GJ79]. We will show that no *NP*-complete problem can have a polynomial time incremental algorithm (with restricted history) unless $P = NP$.

Theorem 3.4.1 *There does not exist an incremental algorithm (with restricted history) with complexity less than $T_{3\text{-}SAT}(\ell)/\ell$ for the 3-SAT problem (where ℓ is the number of clauses).*

Proof We can construct a fast initialization for 3-SAT with ℓ clauses as follows: each clause consists of the literals $x_0 x_0 x_0$; this instance has the satisfying assignment $x_0 = T$. The proof follows immediately by Theorem 3.2.3. ■

Corollary 3.4.2 *Given any problem (function) α in* NP, *if there exists an incremental algorithm ΔA that implements α with restricted history in polynomial time, then* $P =$NP.

Proof If such an incremental algorithm exists, then we can build a polynomial transformation to use it to update 3-SAT instances; by Theorem 3.4.1 the incremental algorithm cannot be more than polynomially faster than the static algorithm; hence 3-SAT must be in P; hence every problem in *NP* must be in P. ■

A statement similar to Corollary 3.4.2 was made in [Che76], although it was not proved.

In the arguments above, the characterization of the lower bound depends strictly upon the function at hand and not upon its implementation. However, the history does not depend just upon the function, but also upon the particular algorithm used to compute that function. As long as we consider only incremental algorithms that

use a history that can be produced in time no more than that required for a fast initialization, i.e., $\leq T_\alpha$, the above theorems continue to hold. But for incremental algorithms that depend upon a history that *cannot* be constructed in the same time as a fast initialization, the argument in Theorem 3.2.3 above does not limit the speed they might attain. We discuss this issue further in §4.3.1. Whether a bound can be found that applies to such incremental algorithms remains open. If no such bound exists, it ought to be possible to find incremental algorithms that run very fast but use a lot of history information; there may be a trade-off here. On the other hand, it seems unlikely that if a lot of information is kept around it will be possible to use it, and keep it up to date, sufficiently fast to result in a speed up. It would also be interesting to determine a set of necessary conditions for a function to have a fast initialization.

Chapter 4

Incremental Relative Lower Bounds

4.1 Overview

In this chapter we present a general method that permits simple proofs of *relative lower bounds* for incremental update algorithms.[1] We apply this method to classify functions by relative lower bounds. Our bounds are relative in the sense that they are proportional to the complexity of the problem at hand; if no lower bound is known for the initial computation, then the *relative* lower bound is not necessarily a lower bound. Our technique encompasses a result of Even and Gazit, who showed for the particular problem of computing all shortest paths in a directed graph that an incremental algorithm can do no better, in the worst case, than recomputing the solution with the new inputs [EG85].

We demonstrate our technique by bounding a number of incremental algorithms drawn from various domains. Our results have interesting implications with respect to the optimality of an incremental algorithm previously developed by Ryder in [Ryd83, RP88]. The proof method and function classification suggest which types of problems are likely to yield good (i.e., of lower complexity) incremental algorithms and which cannot be improved by an incremental approach, in the worst case.

4.2 Incremental Relative Lower Bounds

4.2.1 The Proof Method

Our proof method is general, encompassing a variety of lower bounds proofs. We will provide an intuitive description of the method and then lay it out more formally, first

[1] The results in this chapter were published in [BPR90].

discussing the general approach and then showing specific examples. We will show that for certain functions, the lower bound for any *incremental* algorithm is proportional to the lower bound for computing that function. Thus, we call a bound produced by this method an *incremental relative lower bound*. Refer to Chapter 2 for a definition of the terms and symbols used here.

Definition 4.2.1 Given a function α with time complexity T_α, and ρ a function of ℓ, if we can show that, for any incremental algorithm ΔA for α, $T_{\Delta A}(\ell) \cdot \rho(\ell) \geq T_\alpha(\ell)$, then α has an *incremental relative lower bound (IRLB)* of $1/\rho(\ell)$.

If T_α is known, we can use an IRLB to derive immediately a (non-relative) lower bound for the incremental algorithm. More typically, we have a "best known" algorithm, and our lower bound is relative to the complexity of that algorithm.

The idea of the proof method is to build a (non-incremental) algorithm for α using an arbitrary incremental algorithm as a subroutine. We count the number of times the incremental algorithm is used. Assuming that we can disregard certain additional overhead in the constructed algorithm, which may not always be the case, the number of invocations of the incremental algorithm is the inverse of the IRLB for α. For example, suppose that we can take any input p, do a small amount of preprocessing on it, and then run a single step of an incremental algorithm to compute $\alpha(p)$. How much faster could the incremental algorithm be than an algorithm to compute the entire function? Since an algorithm to compute this function can be constructed using a single call to the incremental algorithm, plus a little extra time for preprocessing, the incremental algorithm cannot be asymptotically faster than the non-incremental algorithm. In general, the relative bound that we can prove will depend upon the amount of time required for preprocessing, and the number of times that the incremental algorithm needs to be called in the constructed algorithm.

Figure 4.1 is a formal description of the procedure described above. In this procedure, we start in steps 1–3 by constructing an easily-solved instance of the problem, by making $\delta(\ell)$ changes. In step 4, we solve the easy instance to compute an initial value in the range of α. For example, consider pairwise connectivity in an undirected

0. input problem $p_{initial}$ of length ℓ;
   /* *change p to a "fast initialization value"* */
1. **for** $i \leftarrow 1$ to $\delta(\ell)$ **do**
2. make an incremental change to p
3. **end**;
   /* *Solve the instance* */
4. $q \leftarrow \alpha(p)$;
   /* *Do any algorithm-dependent preprocessing —*
   h is additional information used by the incremental algorithm */
5. $h \leftarrow$ initial history for p;
   /* *Incrementally change p to get back to an instance with same*
   value in range as original input, using ΔA *to update* $\alpha(p)$ */
6. **for** $i \leftarrow 1$ **to** $\epsilon(\ell)$ **do**
7. $p \leftarrow p$ with one incremental change;
8. $q \leftarrow \Delta A(p)$; /* *assume h updated as side effect* */
9. **end**;
   /* *if at the exit to the loop,* $\alpha(p) = \alpha(p_{initial})$, *then* $q = \alpha(p_{initial})$ */

Figure 4.1: Procedure for IRLB proof

graph — given a graph G and two nodes x and y, is there a path from x to y? If in steps 1–3 we add an artificial node and connect every node in the original graph to that node, then the entire graph is now connected, regardless of the original graph. Thus, in step 4, we know that the value of the function is "True" for every pair in the original graph. This is illustrated for an example graph in Figure 4.2. For this example, step 2 adds elements to the initial input; more generally, an element in the input set might be replaced by a different element, e.g., for some problem the construction of the "easily-solved instance" could require that the weight of a graph edge be changed. Step 5 is included since an incremental algorithm may require additional information such as specialized data structures. Steps 6–9 represent the process of modifying the input $\epsilon(\ell)$ times in order to get to another input that is equivalent in the range to the original input. Continuing the example of graph connectivity, if all the edges to the artificial node are removed, then in the final pass the graph p will have the same connectivity as the original input graph. (Note that it is not necessary to get back to $p_{initial}$ — only that $\alpha(p)$ be equal to $\alpha(p_{initial})$).

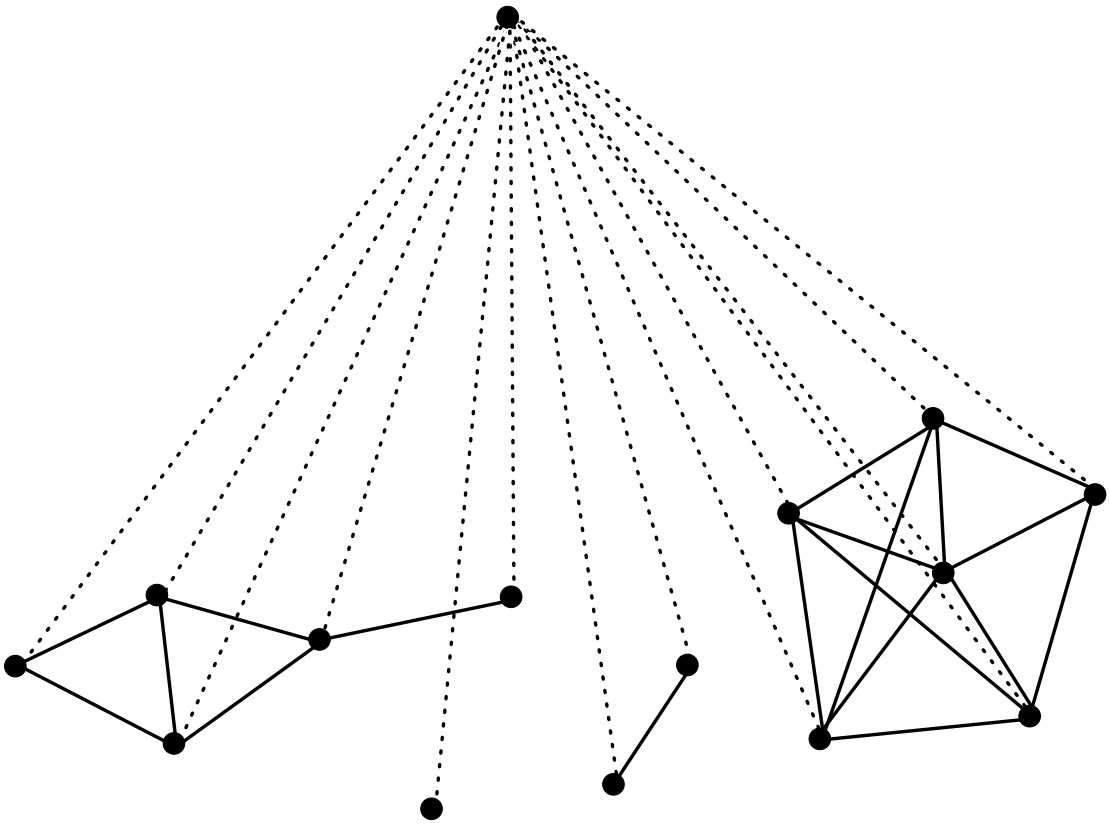

Figure 4.2: Example of proof method: dashed lines added in step 2.

The rest of this section explores the details of our theorem, and its applicability. We assume in this procedure that the steps that add, change, or remove elements in p, steps 2 and 7, are constant-cost operations. The exact bound that can be proved depends on the following factors in the particular instance of the procedure shown in Figure 4.1:

1. the function $\delta(\ell)$ (the number of incremental changes made in the input to get to the easily-solved instance);

2. the cost of step 4 (initialization);

3. the cost of step 5 (preprocessing);

4. the function $\epsilon(\ell)$ (the number of changes made to get to a problem with the same value in the range as $p_{initial}$);

The condition asserted at the end of the procedure does *not* require that $p = p_{initial}$, but only that p and $p_{initial}$ map to the same element of the range of α. Thus in general $\delta(\ell) \neq \epsilon(\ell)$. In order to compute an initial answer quickly in step 4, we typically select a transformation in steps 1–3 in order to obtain a trivially computed problem instance. The significance of the complexity of step 5, preprocessing, is considered in Section 4.3.3. The combined complexity of steps 4 and 5 is denoted $init(\ell)$. We also assume that the complexity of the function, T_α, *dominates* the cost of steps 1 through 5, in the sense of Definition 4.2.2.

Definition 4.2.2 Given two non-negative functions $f(n)$ and $g(n)$, if

$$g(n) = O\left(g(n) - f(n)\right)$$

then we say $g(n)$ *dominates* $f(n)$.

An additional complication arises because the length of p, the problem instance, does not necessarily remain fixed in the construction procedure. In Step 2 of the construction we permit the procedure to add elements to p — we must assure that this change in the problem size does not change the asymptotic lower bound. Let $\ell_{\max}$ be the length of p at step 4. We will assume that our complexity measures behave monotonically, and hence $T_{\Delta A}(\ell_{\max}) \geq T_{\Delta A}(\ell)$. The proof of Theorem 4.2.4 will require that

$$T_{\Delta A}(\ell_{\max}) = \Omega\left(1/\epsilon(\ell)\right) \Rightarrow \; T_{\Delta A}(\ell) = \Omega\left(1/\epsilon(\ell)\right);$$

Lemma 4.2.3 gives sufficient conditions for this implication to hold.

Lemma 4.2.3 *If $T_{\Delta A}(\ell)$ is a polynomial function $f(\ell)$, and $\ell' = c\,\ell$ for some constant c, then:*

$$T_{\Delta A}(\ell') = \Omega\left(g(\ell)\right) \Rightarrow T_{\Delta A}(\ell) = \Omega\left(g(\ell)\right).$$

Proof By the definition of $\Omega\left(g(\ell)\right)$, there exist constants ℓ_0 and k_0 such that for every $\ell' > \ell_0$, $T_{\Delta A}(\ell') \geq k_0\, g(\ell)$. By assumption, $T_{\Delta A}(\ell')$ is a polynomial $f(\ell')$, and $\ell' = c\,\ell$,

so $f(c\,\ell) \geq k_0\, g(\ell)$. Consider $\lim_{\ell\to\infty} \frac{f(c\,\ell)}{f(\ell)}$; this must equal c^k, where k is the degree of f. Furthermore,

$$\lim_{\ell\to\infty} \frac{f(c\,\ell)}{c^k f(\ell)} = 1 \qquad \text{implies that} \qquad \lim_{\ell\to\infty} \frac{f(c\,\ell)}{2c^k f(\ell)} = \frac{1}{2}.$$

Thus for large ℓ, $2c^k f(\ell) > f(c\,\ell)$. Let $k_1 = 2c^k$; then for $\ell > \ell_1$, $k_1 f(\ell) > f(c\,\ell) \geq k_0 g(\ell)$; thus, $f(\ell) > (k_0/k_1)g(\ell)$. Since k_0/k_1 is constant, we can conclude that $f(\ell) = \Omega\,(g(\ell))$. ■

In the case of exponential complexity, Lemma 4.2.3 does not hold, since

$$\lim_{\ell\to\infty} \frac{2^{c\,\ell}}{2^\ell} = \infty \text{ for } c > 1.$$

However, the implication in the lemma can be shown to hold for other common sub-exponential functions, e.g., $f(\ell) = \ell \log \ell$.

Theorem 4.2.4 *Consider an incremental algorithm ΔA for function α and a procedure of the form shown in Figure 4.1 constructed using ΔA. If:*

i. T_α dominates $\delta(\ell) + \text{init}(\ell)$ (steps 4 and 5);

ii. $T_{\Delta A}$ (and hence T_α) is polynomial, and $\ell_{\max} = c\,\ell$; and

iii. when the procedure terminates, $\alpha(p) = \alpha(p_{initial})$

then the function α has an IRLB of $1/\epsilon(\ell)$.

Proof Let $|p_i|$ be the length of the problem instance at the ith iteration of the loop at steps 6–9. Then the complexity of the procedure described above is approximately

$$\ell + \delta(\ell) + init(\ell) + \sum_{i=1}^{\epsilon(\ell)} T_{\Delta A}\,(|p_i|)\,.$$

We can bound the last term by $\epsilon(\ell) T_{\Delta A}(\ell_{\max})$. If condition *iii* is met, then the procedure described computes $\alpha(p)$. We compare the cost of this routine to the cost of the optimal algorithm for α; since the optimal cost, by assumption, omits the input procedure, we omit it from both sides of the equation. Hence by Definition 2.1.1,

$$T_\alpha \leq \delta(\ell) + init(\ell) + \epsilon(\ell) T_{\Delta A}(\ell_{\max}).$$

Solving for $T_{\Delta A}(\ell_{\max})$,

$$T_{\Delta A}(\ell_{\max}) \geq \frac{T_\alpha - \delta(\ell) - init(\ell)}{\epsilon(\ell)}.$$

Since we assume by condition i that T_α dominates the other terms,

$$T_{\Delta A}(\ell_{\max}) = \Omega\left(\frac{T_\alpha}{\epsilon(\ell)}\right).$$

Using condition ii and Lemma 4.2.3, we can conclude that

$$T_{\Delta A}(\ell) = \Omega\left(\frac{T_\alpha}{\epsilon(\ell)}\right);$$

thus, α has an IRLB of $1/\epsilon(\ell)$. ■

4.2.2 Significance of the Proof Conditions

Condition i requires that one carefully assess the costs of initialization ($\delta(\ell) + init(\ell)$) to be certain that they are dominated by T_α. If these conditions cannot be met, it is still possible to apply a weaker theorem. That theorem, described in [PaCB84], yields an IRLB of $1/\ell$, which, of course, is not useful for functions with linear lower bounds.

Condition ii in Theorem 4.2.4 is needed only when the changed problem increases in length. The following Theorem substitutes instead the condition that the size of p is not permitted to change in the algorithm constructed for the proof. With this modification, certain bounds are still easily provable for functions not known to be polynomial — for example, for those that are NP-complete. This theorem appeared with a different proof in [PaCB84].

Theorem 4.2.5 *Consider an incremental algorithm ΔA for function α and a procedure of the form shown in Figure 4.1 constructed using ΔA. If:*

i. T_α dominates $\delta(\ell) + \text{init}(\ell)$ (steps 4 and 5);

ii. when the procedure reaches line 4, $|p| \leq |p_{initial}|$; and

iii. when the procedure terminates, $\alpha(p) = \alpha(p_{initial})$;

then the function α has an IRLB of $1/\epsilon(\ell)$.

Proof The proof is the same as that for Theorem 4.2.4, except that the new condition *ii* implies that $\ell = \ell_{\max}$. Therefore, the last step of the proof — and hence the condition that T_α is polynomial — is not needed. ■

4.3 The IRLB Classification

The problems to which we have applied Theorem 4.2.4 can be divided into three relative complexity classes. In this section, we classify some common problems. We also characterize a general class of functions that are in *Class 1*, using a corollary of Theorem 4.2.4. The IRLB classification is presented in Figure 4.3. Since Class i represents

Class	IRLB	Example
1	$1/\ell$	Sorting
2	$\leq 1/\ell, \geq 1/\sqrt{\ell}$	Minimum Spanning Tree
3	1	All Shortest Paths

Figure 4.3: A Classification of IRLB's

functions with tighter bounds than those in Class $i-1$, a function α in Class i is also in Class $i-1$, but the converse is not true. Unless a function is known to have an incremental algorithm that meets its IRLB, then its placement in the classification is necessarily tentative, since it is always possible that a more sensitive proof technique may yield a tighter bound.

Although the classification does not span all potential IRLB's in the range 1 through $1/\ell$, every problem that we have been able to analyze lies in one of the 3 classes shown. The Class 2 functions arise with graphs; the range of IRLB's comes from the structure of the family of graphs considered as input. In particular, sparse graphs will be at one end of the range $(1/\ell)$ and dense graphs will be at the other end $(1/\sqrt{\ell})$. While there may be functions that fall between Class 2 and Class 3, we are not aware of any.

4.3.1 Sufficient Conditions for a Class 1 IRLB

Is there any function that cannot be proved to have an IRLB, that is, which is not in Class 1? We do not have an example, but it is not obvious such a function does not exist. We can show that the existence of a *fast initialization value* (Definition 3.2.1) is *sufficient* to prove an IRLB, but we do not know if it is necessary. Since a fast initialization can be found for most problems encountered in practice, the following corollary to Theorem 4.2.4 is quite general. Note that this corollary is a restatement, and re-proving, of Theorem 3.2.3 in terms of the IRLB.

Corollary 4.3.1 *If a function α has a fast initialization value, then it has an IRLB of $1/\ell$. [PaCB84]*

Proof We prove this by applying Theorem 4.2.5. To do this, we must first describe a procedure of the form shown in Figure 4.1, and then prove that this procedure meets the preconditions for the theorem.

The Procedure: In steps 1 through 3, we incrementally change each element in the input problem to an element in the fast initialization value; thus we make ℓ iterations, and exit with p being the fast initialization value of the same length as the original input. In step 4, q can be computed quickly, by the definition of a fast initialization; without loss of generality, assume that no history is produced in step 5. In the loop starting at step 6, the procedure restores the original input, incrementally; this requires ℓ iterations. Hence when the loop is exited, $p = p_{initial}$ and thus $\alpha(p) = \alpha(p_{initial})$.

The Preconditions: The number of iterations of the loop in steps 1–3, $\delta(\ell)$, is ℓ; steps 4 and 5 are fast by assumption; hence T_α dominates these terms, and precondition *i* can be applied. Since our incremental changes leave p at the original size, precondition *ii* holds. The third precondition holds since $p = p_{initial}$ at the end. Hence, Theorem 4.2.5 can be applied with $\epsilon(\ell) = \ell$, deriving an IRLB of $1/\ell$. ■

4.3.2 Examples of Class 1 Functions

Class 1 problems are common. For example, suppose our input is a set of integers, and the function returns the set ordered from lowest to highest. We know that sorting has an IRLB of $1/\ell$ since it has a fast initialization value $(1, \ldots, n)$. Since sorting by comparisons has a proven lower bound of $\Omega(\ell \log \ell)$, incremental sorting by comparisons must have a lower bound of $\Omega(\log \ell)$.

Corollary 3.4.2 in §3.4 demonstrated, in effect, that 3-SAT has an IRLB of $1/\ell$, and hence no *NP*-complete problem can have a polynomial time incremental algorithm unless $P = NP$.

We have not found any functions for which we could prove that no fast initialization exists; this is an interesting open problem. Of course, even in the absence of a fast initialization, it may be that a function falls into Class 1. However, without a fast initialization (and thus Corollary 4.3.1) it is not obvious how to prove an IRLB.

4.3.3 Fast Initializations and Preprocessing: Some Observations

We now present an example that demonstrates the need to be careful about fast initializations and preprocessing time. Consider the following familiar pattern matching problem:

> S is a string of n symbols, called the *subject*, and P is a string of m symbols, called the *pattern*. MATCH(P,S) is *True* if one or more occurrences of P are in S, and *False* otherwise. The particular incremental variation we will consider will permit changes in S but not in P.

We might make the following argument: MATCH has a simple fast initialization; simply change the first m symbols of S to P, and answer *True*. This requires m steps. Then S can be restored to its original state via m steps of an incremental algorithm for MATCH. Hence we conclude that MATCH has an IRLB of $1/m$. Assuming that $T_{\text{MATCH}} = \Theta(n)$, this argument suggests that no incremental algorithm for MATCH can be faster than $\Theta(n/m)$.

On the other hand, suppose we construct an incremental algorithm that keeps track of the position of each occurrence of P within S, in a table, plus a count of the number

of matches. It is easy to show that each time a change occurs in S, the algorithm need only look in the "neighborhood" of the change, to see if the table and count need to be updated. When the count goes to 0, the answer becomes *False*, and conversely when the count increases from 0, the answer becomes *True*. This is an incremental algorithm for the problem above, requiring $\Theta(m)$ time. Since in a "typical" problem instance, $m < \sqrt{n}$, this incremental algorithm is faster than our reputed lower bound. What gives?

In fact, there is no contradiction, since the initialization described in the IRLB argument does not build a table of the positions of P in S; it never even considers S at all. By ignoring the preprocessing (step 4 in the procedure) we miss the fact that the total initialization time is not dominated by the complexity of the algorithm. If the initialization did examine S, it would require time $\Theta(n+m)$, and would no longer be a fast initialization. This example illustrates the importance of the preconditions on the proof, and implies a trade-off between initialization and preprocessing on the one hand, and fast incremental updates on the other.

4.4 Incremental Relative Lower Bounds for Graph Problems

In this section we present some graph problems and argue their IRLB classification.

4.4.1 A Recapitulation of All-pairs Shortest Paths

Even and Gazit show that no incremental algorithm for the All-pairs Shortest Paths (ASP) problem in a directed graph can be any faster, in the worst case, than recomputing from scratch [EG85]. In our terms, this places the function ASP in *Class 3*. Here we recapitulate Even and Gazit's argument to show how it fits within our framework.

The key step in demonstrating the IRLB is to exhibit an algorithm that implements the function at hand, using an incremental update algorithm for the same function as a subroutine. Given an input graph for the shortest path problem, $G = (V, E)$, modify it by adding two vertices, v^* and v^{**}. For each vertex $v_i \in V$, add edges $v_i v^*, v^{**} v_i$ of weight 0; also add edge $v^* v^{**}$ of weight 0. This construction is illustrated in Figure 4.4.

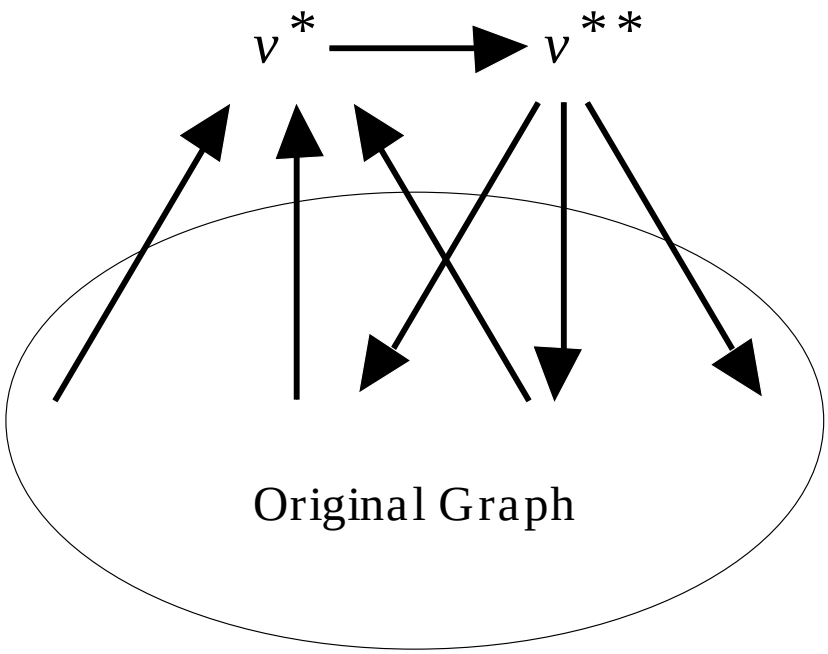

Figure 4.4: All Shortest Paths proof construction

This construction corresponds to Steps 1 and 2 in the procedure shown in Figure 4.1, and can be done in $\Theta(n)$ steps, where n is the number of vertices in G; since $n \leq \ell$, Steps 1 and 2 are $O(\ell)$. Step 3, the initialization, is immediate, since one shortest path between any two vertices a and b in the constructed graph is $a\, v^*\, v^{**}\, b$ with length 0. Step 4 can be performed in parallel with 0, 1, and 2 in Even and Gazit's algorithm. To get back to a graph that has the same solution as the original input merely requires the removal of a single edge, v^*v^{**}. The loop function in Step 6, $\epsilon(\ell)$, is 1. Thus we can conclude immediately that ASP has an IRLB of 1, and is in *Class 3*.

4.4.2 Graph Connectivity Problems

The connected components function takes as input an undirected graph $G = (V, E)$ and outputs one or more lists of vertices such that if v_i and v_j are in the same list, then there exists a path between them. We construct an initial trivial solution by forcing all vertices to be in a single component as follows: add an artificial vertex v^*, and for each $v_i \in V$, add an edge v_iv^*. In terms of Figure 4.1, $\delta(\ell) = n$ where $n = |V|$. Since

in any representation of G, $\ell \geq n$, Steps 1 and 2 meet the conditions for the theorem. Step 4 is immediate: all vertices are in a single component. To return the problem to its original solution, each of the edges added in Step 2 must be removed, with the incremental algorithm after each removal. Thus $\epsilon(\ell) = n$ is bounded above by ℓ when the graph is sparse, that is, $|E| = O(n)$, and below by $\sqrt{\ell}$ when $|E| = \Omega(n^2)$. Connected components is in *Class 2*.

For a undirected graph, two vertices are in the same biconnected component if there exist two paths between the vertices with no common intermediate vertices. The proof for biconnected components uses a construction similar to that in Figure 4.4. We add two vertices v^* and v^{**}, and connect each old vertex in the graph to each of these new vertices. Now each pair of old vertices v_i and v_j have two distinct paths between them, so the entire graph is biconnected. To transform this graph to one with the same biconnected components as the original graph, we must remove all the added edges $v_i v^*$ and $v_i v^{**}$, a total of $2n$ changes. Since 2 is a multiplicative constant, the same bound applies as in the connected components case and this function is in *Class 2*.

In a weighted graph, each edge $v_i v_j \in E$ has a number $w(v_i, v_j)$ associated with it, called the *edge weight*. The *minimal spanning tree* problem is to find a set of edges that connect E such that the sum of the edge weights is minimized. A construction similar to the two above can be used. A new vertex v^* is added, with an edge between v^* and every vertex in the original graph. For each added edge $v^* v_i$, $w(v^*, v_i) = -\infty$. The initial solution to the modified graph can be found by selecting the set of edges from v^* to the original vertices, which connects the tree and certainly has minimal weight. To transform this graph back to the original graph, the n added edges are removed; hence $\epsilon(\ell) = n$, and minimal spanning tree is in *Class 2*.

4.5 Solving Systems of Equations

This section considers problems formulated as a system of equations whose the solution yields the problem solution [Pau88]. We demonstrate some common problems that can

be given this representation.[2] Our systems of equations are either *unconstrained* or *constrained*, depending on the relation between the constants and coefficients. We first consider unconstrained systems.

A problem instance of a function α consists of a set of coefficients and constants drawn from main P, defining a system of equations (4.5.1).

$$X_i = \sum_{k \in L_i} a_{i_k} \star X_k + c_i, \text{ for } 1 \leq i \leq n, \text{ where } L_i \subseteq \{1, \ldots, n\} \tag{4.5.1}$$

In this context, A, an algorithm that implements α, is a solution procedure for that system of equations.

Let $\sum_{i=1}^{n} |L_i| = p$. Then there are $p + n$ elements in the input: the a_{i_k} and c_i. The binary operators $+$ and $\star$ must satisfy the following properties:

1. There exists an identity element, 0_+, with respect to $+$: $x + 0_+ = x, \forall x \in P$.
2. There is an annihilator, 1_+, with respect to $+$: $x + 1_+ = 1_+, \forall x \in P$.
3. There is an identity element, $1_\star$, with respect to $\star$: $1_\star \star x = x, \forall x \in P$.[3]

Certain classical data flow analysis problems, such as available expressions and very busy expressions, can be characterized in this manner. With some additional restrictions on the form of the system, the reaching definitions and connected components problems can be modeled as well. For an overview of data flow analysis problems, see [ASU86]; for a discussion of the system equations model applied to data flow analysis, see [Hec77] and [RP86].

A system of equations of form (4.5.1), can be modeled as a digraph constructed from the variable interdependencies. Each variable corresponds to a node in the digraph; the appearance of X_j on the right hand side of the equation for X_i means that an edge between j and i appears in the digraph. The direction of the edges can be all from j to i or all from i to j, whichever is natural for a particular problem. We can think of

---

[2] Our discussions are in terms of systems in which the equations are linear, but the methods generalize to non-linear systems as well.

[3] Actually, a weaker property is sufficient for our purposes: that $1_\star$ be an identity for 1_+ and 0_+. The more general assumption simplifies our explication.

a_{i_j}, the coefficient for X_j, as the label on the edge j, i. The constant term for X_i, c_i, is a label associated with the node. Thus, we can speak of changes to the system of equations as changes to this digraph model.

4.5.1 IRLB's for Unconstrained Systems of Equations

Recall that an IRLB proof requires a way to get from the original problem instance to a special instance that can be easily solved, followed by a sequence of incremental steps leading back to a solution of the original problem. Given an original problem of the form (4.5.1), construct the equations (4.5.2), by adding the $1_\star \star X_{n+1}$ term to each of the original equations, and add a new equation (4.5.3).

$$X_i = \sum_{k \in L_i} a_{i_k} \star X_k + 1_\star \star X_{n+1} + c_i \quad \text{for } 1 \leq i \leq n \tag{4.5.2}$$

$$X_{n+1} = c_{n+1} \tag{4.5.3}$$

A digraph representation of the above transformation is shown in Figure 4.5.

Assume that algorithm A solves α, that is, it solves a system of equations (4.5.1).[4] Refer back to Figure 4.1. Adding the n coefficients and one constant of (4.5.2) and (4.5.3) corresponds to steps 1–3 of the procedure. Examination of these modified equations reveals that by letting $c_{n+1} = 1_+$, the system yields the trivial solution $X_i = 1_+$, $1 \leq i \leq n+1$. Therefore step 4 is accomplished at constant cost. By changing c_{n+1} to 0_+, we perform steps 6–9 with $\epsilon(\ell) = 1$. Note that a solution to this system is a solution to the original system. Thus by Theorem 4.2.4, any function that can be modeled as a system of equations (4.5.1), with the conditions on $+$ and $\star$ stated above, is a Class 3 function, since $\epsilon(\ell) = 1$ in this construction.

An example of a function whose solution can be modeled as above is the minimum-maximum edge weight path, or bottleneck flow problem, as described in [Pau88]. Consider a weighted digraph with a distinguished node v. For each node i in the graph, the goal is to find a route that minimizes the weight of the most costly edge in the path

[4] The phrase "solves a system of equations" should not be construed to impose any particular algorithm upon the solution. The only requirement is that a value has been determined for each of the X_i, by any means.

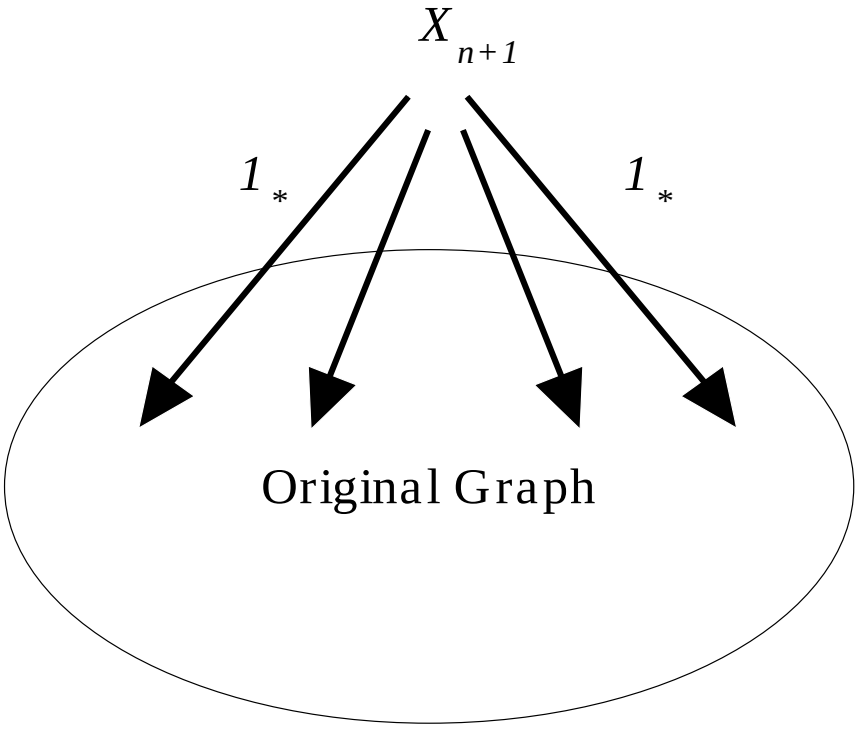

Figure 4.5: Additions to (4.5.1) for IRLB proof

between i and v. This problem can be formulated as a set of equations (4.5.1) where $\star$ is *maximum* and $+$ is *minimum*. For this problem the domain is the non-negative integers, with $0_+ = \infty$ — or any number larger than the maximum edge weight in the particular problem instance — and $1_+ = 0$. Then X_i is the maximal weight on a path from i to v; each a_{i_k} is the weight on edge i, j; and each $c_i = \infty$.

To demonstrate the IRLB argument for this function, add a new node X_{n+1}, with an edge $i, n+1$, weighted 0, for every node i in the original graph. This corresponds to adding a term $0 \star X_{n+1}$ to every equation X_i. Then set the node weight for X_{n+1} to 0, corresponding to equation (4.5.3) above. This set of equations has a "trivial" solution $X_i = 0$ for every i. Then, change the value of X_{n+1} to ∞; since $+$ corresponds to minimum, none of the added terms can have any effect, so a single call to an incremental algorithm yields a solution to the original set of equations.

4.5.2 IRLB's for Constrained Systems of Equations

Now consider problems modeled as sets of equations (4.5.1) where a_{i_k} and c_i are interdependent: $c_i = f_i(a_{i_1}, \ldots, a_{i_n})$. Under this constraint, we cannot change an a_{i_k} without potentially changing c_i — thus the change of an a_{i_k} and the resulting change of c_i is considered a unit change to the input.[5] We can apply our IRLB methods to this system by imposing some new conditions; depending on the conditions chosen, the associated functions will fall either in IRLB Class 2 or IRLB Class 3.

An example of a constrained system of equations can be derived from the reaching definitions problem of data flow analysis [Hec77, RP86]. A flow graph is a digraph representation of the execution flow in a single-entry procedure. Each node represents a straight-line code sequence; edges represent possible transfer of control at execution time. We refer to a variable in the procedure being analyzed as a *location* and to a variable in the system of equations as *variable* in the discussion below. A *definition* of location x at node i is a statement in the code associated with node i that (potentially) changes the value of x, e.g., $x \leftarrow 1$. If a node i does not contain any definitions for location x, then we say that i *preserves* definitions of x. If there is a definition of x in i, and a path through the flow graph from i to j such that all intervening nodes preserve x, we say that definition (x, i) *reaches* node j. That is, the value given to location x in node i may still be contained in x when entering node j. The *reaching definitions problem* is to find the set of reaching definitions for each node in the flow graph.

We discuss first the IRLB proof for reaching definitions in terms of the digraph model; then we will show how this proof can be generalized for systems of equations that correspond to the same general conditions as this problem. The proof construction is equivalent to adding two nodes to the flow graph — see Figure 4.6. Note that the nodes $1, \ldots, n$ are shown twice in the diagram to simplify the construction. Node $n+2$ is the immediate predecessor of every other node in the flow graph; we set it to preserve all location definitions that it receives. Node $n+1$ is the successor of all nodes, and the predecessor of $n+2$. We set it to preserve all location definitions that reach it.

[5] A change to a single c_i could require changes to more than one of the a_{i_k} and hence would be considered a sequence of changes rather than a single change; see Chapter 2.

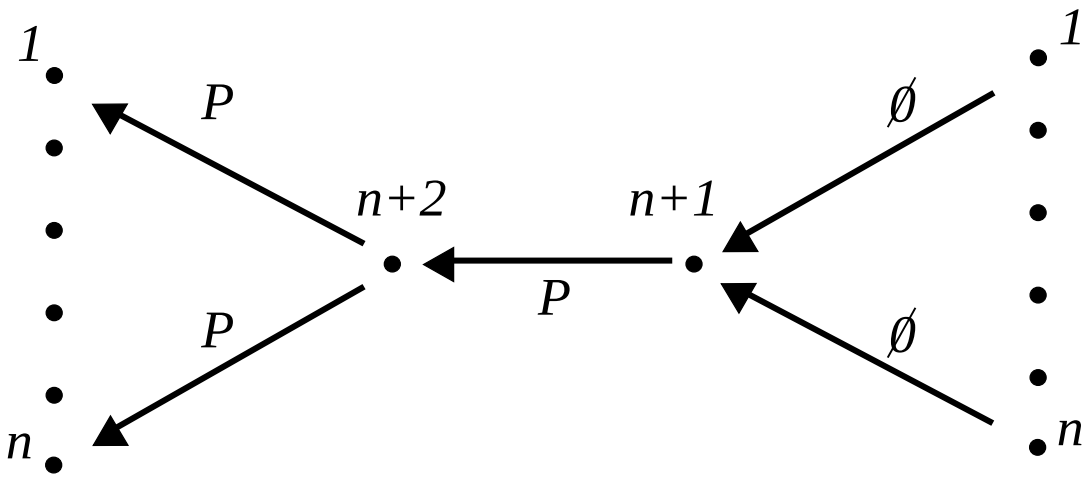

Figure 4.6: Additions to Constrained System for Class 3 IRLB Proof

This process of setting up the new nodes and edges corresponds to Steps 1 through 3 in the IRLB proof procedure; $\delta(\ell) = n + 1$. Examination of the digraph reveals that the reaching definitions set for each node is the set of all location definitions in the program — this is our "trivial solution" for step 4. Now the incremental change is made: remove the edge from $n + 1$ to $n + 2$. Since $n + 2$ preserves all location definitions, and it now has no definitions upon entry, it can have no effect on the nodes in the original digraph. Hence, an algorithm processing this single incremental change will yield a solution to the original problem. Since a single call is made to the incremental algorithm, $\epsilon(\ell) = 1$, and the reaching definitions problem is in Class 3.

To model this problem as a system of equations, we associate a variable with each node in the flow graph. X_i is the set of location definitions in the procedure that reach the entry to node i on some path from the procedure entry node. We also associate with node i the set of the new definitions associated with i; these are called the *last definitions* for i. Then c_i is the union of the sets of last definitions of locations associated with immediate predecessors of node i in the flow graph. The a_{i_k} are sets of definitions preserved by node k; thus, if P is the union of all definitions of program locations in the procedure, then $a_{i_k} = P$ minus all definitions of locations that are in the last definitions for node k. Consequently, c_i and the a_{i_k} are interdependent. For the reaching definitions system, $\star$ is $\cap$ and $+$ is $\cup$.

The proof for reaching definitions depended implicitly on the following conditions on the system of equations:

1. There exist $a_{i_1}, \ldots, a_{i_n}$ such that $f_i(a_{i_1}, \ldots, a_{i_n}) = 1_+$ (thus allowing $c_i = 1_+$.)
2. There exists an element $0_\star \in P$ such that $0_\star \star x = 0_\star$, $\forall x \in P$ — that is, $0_\star$ is an annihilator for $\star$.
3. $f_i(1_\star) = 0_+$; and $f_i(a_{i_1}, \ldots, a_{i_n}, 1_\star) = f_i(a_{i_1}, \ldots, a_{i_n})$

Our IRLB proof construction is based on the following modification to the original set of equations:

$$X_i = \sum_{k \in L_i} a_{i_k} \star X_k + 1_\star \star X_{n+2} + c_i, \quad \text{for } 1 \leq i \leq n \tag{4.5.4}$$

$$X_{n+1} = \sum_{k=1}^{n} a_{(n+1)_k} \star X_k + c_{n+1} \tag{4.5.5}$$

$$X_{n+2} = a_{(n+2)_{n+1}} \star X_{n+1} + c_{n+2} \tag{4.5.6}$$

where $a_{(n+1)_i}$ are chosen so that $c_{n+1} = 1_+$. Conditions 1 and 3 above allow us to build this new system of equations from the original system. Note that two equations are added (4.5.5, 4.5.6), plus additional terms added to the X_i's in the form shown by (4.5.4). A digraph representation for such a system is shown in Figure 4.6.

To find the initial "trivial" solution required by the proof, let $a_{(n+2)_{n+1}} = 1_\star$ (and hence $c_{n+2} = 0_+$). Since $X_{n+1} = 1_+$ (1_+ annihilates all the $a_{(n+1)_k}$'s), $X_{n+2} = 1_\star \star 1_+ + 0_+ = 1_+$. Now, each equation X_i has a term $1_\star \star 1_+ = 1_+$, and since the 1_+ term annihilates the other terms, the system has the solution $X_i = 1_+, \forall i$. This corresponds to steps 1 through 4 of the procedure. Next, the system is modified incrementally by letting $a_{(n+2)_{n+1}} = 0_\star$ — which makes $X_{n+2} = 0_\star \star 1_+ + 0_+ = 0_+$; then, the incremental algorithm is applied. Each of the X_i equations now has a term $1_\star \star 0_+ = 0_+$; hence this single change yields a system with the same solution as the original system. It follows that the number of calls to the incremental algorithm, $\epsilon(\ell)$, is 1. Therefore, by Theorem 4.2.4, Class 3 contains all systems of this form.

4.5.3 Constrained systems in IRLB Class 2

Problems that correspond to unweighted, undirected graphs are modeled by sets of equations that have additional constraints. For example, since the edges are not directed, the coefficients are symmetric, i.e., $a_{i_k} = a_{k_i}$. Such a set of equations can be used to solve the connected components problem, which was shown in Section 4.4.2 to be in Class 2. It can be shown that any problem that can be modeled in this way is in Class 2; the details can be found in [BPR86].

4.6 Implications

4.6.1 A Collection of IRLB's

Figure 4.7 lists certain common functions for which we have proven IRLB's better than $1/\ell$. Some of the arguments appear in this paper; the others can be derived easily.

Problems with O(1) IRLB's (Class 3)	**Reference**
Shortest path in a digraph (various forms)	§ 4.4.1, [AHU74]
Transitive Closure of a binary relation	§ 6.3
Planarity Testing	[AHU74]
Strong Connectivity	[AHU74]
Minimum-Maximum edge weight path	§ 4.5.1, [Pau88]
Reaching Definitions	§ 4.5.2, [RP86]
Available Expressions	[ASU86]
Live Uses of Variables	[ASU86]
Domination	[Hec77, RP86]

Problems with $\leq 1/\ell,\ \geq 1/\sqrt{\ell}$ IRLB's (Class 2)	**Reference**
Connected Components	§ 4.4.2
Biconnected Components	§ 4.4.2
Minimum Spanning Tree	§ 4.4.2
Shortest path, undirected graph	[AHU74]

Figure 4.7: Categorization of Sample Problems

4.6.2 An Incremental Algorithm for Minimal Spanning Trees

Frederickson has presented an algorithm for incremental update of edges in a Minimal Spanning Tree (MST), and an extension that can be used to update the Connected Components in a graph [Fre85]. His algorithm runs in time $O(\sqrt{m})$, where m is the

number of edges in the graph. The algorithm requires $O(m)$ preprocessing, plus the time to find the initial minimum spanning tree, which is known to be $O(m \log \beta(m,n))$[6] [GGST86].

In Section 4.4.2, we described a method for demonstrating that MST is in IRLB Class 2. Implicit in this proof is that Step 5 in the IRLB proof procedure (Figure 4.1), which corresponds to the preprocessing phase in an incremental update algorithm, must be performed in time *dominated*[7] by T_α, the complexity of computing the function from scratch. Certainly, MST has a lower bound of $m + n$, where m is the number of edges and n the number of vertices. Thus, our proof does apply to Frederickson's algorithm, since the $O(m)$ time required for preprocessing[8] is dominated by the lower bound for MST. For our purposes we will refer to a set of MST problems as *dense* if $m = kn^2$, $0 < k \leq 1$. For such a set of problems, the length of the problem, ℓ, is proportional to n^2, and we can derive an incremental lower bound of $\ell/n = n^2/n = n = \sqrt{m}$. Thus for dense graph sets, the complexity of Frederickson's method is optimal to within a constant factor. This suggests that if the number of edges grows more slowly than n^2, there may be some improvement possible over Frederickson's result. This may occur for example if m is bounded by some constant times n. An identical argument to this can be made for connected components.

4.6.3 Implications for Incremental Data Flow Analysis

The classical data flow analysis problems, such as reaching definitions, available expressions, and very busy expressions, can all be formulated as systems of equations [Hec77, RP86]. By the arguments in Section 4.5, they can be shown to be in *Class 3*. This implies that any incremental algorithm for these problems can have no better worst case performance than the best known non-incremental algorithms.

---

[6] $\beta(m,n) = \min\{i | \log^{(i)} n \leq m/n\}$

[7] See Definition 4.2.2.

[8] Note that the preprocessing in the IRLB proof procedure, which is $O(m)$ plus the time to find the initial tree, is $O(m)$ in the IRLB procedure, rather than $O(m \log \beta(m,n))$, because we have found a fast initialization value for the tree in steps 1–3.

In previous work on incremental algorithms for data flow analysis, Ryder distinguished between two types of incremental changes to these inputs. One type of change occurs because code within a node of the flow graph changes, affecting the a_{ik} or c_i. The other type of change also corresponds to code changes, but the structure of the flow graph is affected; that is, an edge or node is added to or deleted from the graph. We refer to the latter changes as "structural changes." Ryder's original incremental data flow analysis algorithms handled only non-structural changes in code; however, they were extended to handle structural changes later [Bur90, CR88]. Our results show that their worst case behavior can be no better than the best known data flow analysis algorithm. This justifies Ryder's claim that worst case analysis is not an appropriate measure of the utility of these incremental algorithms. It also justifies the alternative performance analyses on a reasonable, structured programming language [RP88].

An interesting question under investigation is the possibility that incremental algorithms which handle only certain classes of code changes, or which are applicable to a restricted set of a_{ik}, c_i, may have better worst case performance than suggested by our IRLB results. By restricting Allen/Cocke interval analysis, on which Ryder's incremental data flow algorithms are based, to programs whose loop nesting level is bounded by a constant, we can improve the worst case bound from $O(n^2)$ to $O(n)$ [RP88]. Similar results may be achievable with incremental data flow algorithms.

Chapter 5

δ-Analysis of Incremental Algorithms

5.1 Overview

In [RR91], Ramalingam and Reps argue that classifying incremental algorithms by their worst-case asymptotic performance may be unnecessarily pessimistic in many cases. As an alternative, they suggest reporting the cost of the update as a function of the size of the *changes* required by the update; this approach is referred to here as δ-analysis, and algorithms that are efficient in terms of δ-analysis are referred to as δ-incremental algorithms.[1] This is a generalization of an idea that first appeared in a paper by Reps [Rep82] and was extended by Alpern et al. [AHR+90]. Reps presented an incremental algorithm for updating an attribute graph with performance linear in the number of vertices in the graph whose attributes are altered by the update.[2] Using this approach along with ideas in [AHR+90], Ramalingam and Reps developed several δ-incremental algorithms on graphs. They also applied a technique from [AHR+90] to prove incremental lower bounds in a restricted model of computation. This chapter will discuss δ-analysis, particularly as a tool for proving lower bounds, and present new lower bounds results. In Chapter 6 we will develop a new δ-incremental algorithm. We discuss the value of this type of analysis versus the alternatives in Chapter 7.

---

[1] In [AHR+90], the term *incremental algorithm* is used as we use δ-incremental algorithm. Unfortunately, the term "incremental algorithm" has been used to mean so many things already that we do not believe that it is sufficiently descriptive of the context, hence δ-incremental.

[2] More precisely, linear in the size of the neighborhood of the change, as discussed in Section 5.2.

5.2 The δ-Analysis Model

The δ-analysis model as developed in [RR91] applies specifically to graph problems. In particular, they consider directed graphs in which there is a value associated with each vertex. Generally the value at a given vertex is computed as a function of the values of its predecessors and/or successors in the graph, i.e., a set of equations where the variables are the values at predecessors and/or successors. A solution consists of some particular fixed point for the set of equations at the vertices. The value at each vertex once a solution has been found is called its *output value*. When a change is made to the structure of the graph, some of the equations will be modified, potentially resulting in new output values at some or all of the vertices.

Because the size of the equation for any given vertex is based on the number of successors and/or predecessors, the actual amount of work required to update a single vertex v is dependent on the number of neighbors (i.e., predecessors and successors) in the graph. Thus the subgraph consisting of v, its neighbors, and the edges connecting them, is called the *neighborhood* of v, denoted $N(v)$. Given a graph $G = (V, E)$, and a subset $K \subseteq V$, the subgraph induced by K is $G_K = (K, E_K)$ where $uv \in E_K$ if and only if $u \in K$ and $v \in K$. Then by extension, the neighborhood of G_K is

$$N(G_K) = G_K \cup \bigcup_{v \in K} N(v).$$

Let the size of the neighborhood be denoted $\|K\|$, also called the *extended size of* K. For some problems it is useful to expand the extended size to the second order, that is, to count the neighbors of K plus their neighbors — the size of this set is the *extended size of* K *of order* 2, denoted $\|K\|_2$.

In δ-analysis, performance is measured as a function of the size of the change in a graph after an update. Consider a change to a single vertex or edge; this will potentially affect the equations for vertices incident on the change. This set of vertices is referred to as the MODIFIED set. We limit our consideration to changes of a single edge; hence |MODIFIED| is always 2. The effect of this change can propagate, resulting in changes to the output values for some of the vertices; the set of vertices that get new output values is called the AFFECTED set. The total set of changed elements is CHANGED =

AFFECTED ∪ MODIFIED, which we refer to as δ.[3] The best δ-incremental algorithms known all depend not strictly on $|\delta|$, but rather on the *extended* size of δ, $\|\delta\|$.[4]

The performance of an incremental algorithm is said to be *bounded* if it is possible to state a bound on its running time as a function of $\|\delta\|$. For example, Reps' algorithm for incremental semantic analysis is $O(\|\delta\|)$. If the running time can be arbitrarily large for fixed $\|\delta\|$, the algorithm is *unbounded*; a problem with no bounded incremental algorithm can be called *non-δ-incremental*.[5]

5.2.1 Thoughts on the Extended Size of δ

It has been stated in previous work that δ-incremental algorithms have the desirable property that the cost of the update depends only on the size of the incremental change, and not on the size of the input. Alpern et al. claim that "... incremental analysis ... studies running time as a function of the size of a *change* in the underlying data structures ...." In [RR91], Ramalingam and Reps state that

> ... instead of analyzing the behavior of algorithms in terms of the size of the size of the *entire* current input ... we concentrate on analyzing algorithms in terms of the size of an "adaptive" parameter ... that captures the changes in the input and output.

However, in a strict sense this is a little misleading. The problem is that it is not really the size of the change, but rather the size of the *neighborhood* of the change that matters, and in general the size of the neighborhood can be as large as n, the number of vertices in the graph. Thus the real cost of a δ-incremental algorithm may depend on the size of the change *and the size of the graph.*

There are a couple of mitigating factors that make this more of a theoretical concern than a practical problem. First, if the graph under consideration is of bounded degree,

---

[3] In his seminal work on δ-analysis [Rep82], Reps used "INFLUENCED" to represent the neighbors of MODIFIED ∪ AFFECTED. We follow the more recent usage in Alpern et al. [AHR+90] and Ramalingam and Reps [RR91].

[4] As pointed out in [RR91], for bounded degree graphs there is a difference of only a constant factor between the size of δ and the size of its neighborhood, for incremental algorithms that depend on a polynomial function of $\|\delta\|$.

[5] Ramalingam and Reps denote such a problem *non-incremental.*

then the size of the neighborhood is bounded by a constant multiple of the size of the change — if the degree bound is d, then $\|\delta\| \leq d \times |\delta|$, which does not depend on n. Second, even if there is no fixed bound on the degree, a low average degree will assure that the size of a typical neighborhood does not grow with n. Unfortunately, it is difficult to predict, in advance, how much of a problem the size of the neighborhood presents. Empirical examination of real data may be the only way to determine whether or not a δ-incremental algorithm is useful for a particular domain.

We might ask, "Is it really 'fair' to count the extended size when computing δ-incremental bounds? Perhaps we should state the algorithm only in terms of $|\delta|$." To see the justification for the extended size, note that the problems under consideration depend upon the computation of a value at each vertex which is some function of the values at the predecessors and/or successors of that vertex. After an incremental change, any but the most trivial function will require recomputing this function by considering the value using each of the arguments. Since the number of arguments depends on the number of neighbors, we are unlikely to do better. While it is not completely obvious that this argument holds in the same way when extending it to sets of vertices — after all, there might be a way to reuse partial answers or compute values for an entire region of the subgraph — it seems clear that any straightforward approach to updating the values will require examination of the predecessors or successors of every vertex. So while it is impossible to dismiss completely the possibility of doing better than $\|\delta\|$, the definition seems compelling for a large class of algorithms. Even trickier issues about the definition of $\|\delta\|$ will arise in Chapter 6.

5.3 Lower Bounds Arguments

5.3.1 Local Persistence

The lower bound argument in [AHR+90] applies to a model of computation termed *local persistence*. In this model, all data is associated with the vertices of the graph. Each vertex is allowed to maintain the following information:

- A list of pointers to its neighbors;

- the formula for the output function;
- the current value of the output function;
- any auxiliary information, *except* that pointers to nodes other than the neighbors are not allowed.

No global auxiliary information is allowed to be maintained in this model.

A locally persistent algorithm must begin processing an update at the point of the change, and execute by following pointers through the graph. At each vertex, the choice of which pointer to follow can depend only on information accumulated in the current pass through the graph — no global auxiliary information from previous passes has been maintained. As each vertex is visited, the local information for that vertex can be updated. All operations are deterministic.

5.3.2 The Incremental Circuit Value Problem

Alpern et al. proved a lower bound for the incremental circuit value problem, within the model of local persistence [AHR+90]. The incremental circuit value problem requires the maintenance of the values computed by gates in a circuit as changes are made to the circuit. The circuit can be modeled by a directed acyclic graph (DAG), where the gates correspond to vertices, the connections between the gates correspond to edges, and the values computed by the gates are the output values. The incremental circuit value problem models applications in spreadsheets and programming environments. The incremental changes allowed are adding a new vertex, deleting a vertex with no edges, adding and deleting edges, and modifying the output function. In the *unrestricted* version of the problem, adding an edge may induce a cycle that must be detected incrementally, whereas the *restricted* version of the problem allows only changes that do not induce cycles; the lower bound applies to both.

The lower bound for the circuit value problem is $\Omega(2^{\|\delta\|})$; an outline of the proof follows. Observe that if any bounded locally persistent algorithm exists, then when an incremental change results in an update that only affects a constant number of vertices in the DAG, the work performed by the algorithm must also be bounded by a constant.

For instance, for all updates that affect only a single node, there must be some constant c such that the algorithm examines no more than c vertices — otherwise, it would be incorrect to say that the algorithm is bounded. In general, for any change of size δ, a bounded update algorithm must have associated with it some function $f(\delta)$ that bounds the number of vertices examined by the algorithm. The proof is then constructed by presenting a particular configuration of the circuit value problem (a type of binary tree), and a sequence of two incremental changes. The first incremental change, which occurs at a leaf, affects a constant number of vertices; such a change can result in a constant bound on the number of vertices any bounded algorithm can visit. The second change, at the root, affects all the vertices on some path in the tree from the root to a leaf; which path in particular depends upon the location of the first change. However, because the first change has only updated a constant number of vertices, the algorithm may have to search the entire breadth of the tree in order to determine which path it needs to update. For a path from the root to a leaf in a binary tree, $\delta = O(\log_2 n)$, while the breadth of the tree is $O(n)$, hence the number of vertices visited by the algorithm must be exponential in δ. The result is an $\Omega(2^{||\delta||})$ lower bound.

5.3.3 Other prior lower bounds

The fact that the bound for the circuit value problem is exponential appears to depend on the acyclic nature of the graph before the change — since the graph is acyclic, the "most costly" change can be found when the graph is structured as a tree. For several other problems in graphs with cycles, it is shown in [RR91] that no bounded locally persistent algorithm can be found; that is, a change affecting only a constant number of vertices may require the examination of every vertex in the graph. These problems are dubbed "non-incremental" by Ramalingam and Reps, although as they point out this claim is only proven in the local persistence model. We refer to them here as non-δ-incremental.

Two methods are used in [RR91] to classify problems as non-δ-incremental. The first is a direct proof similar in form to that used in [AHR+90] — a pair of changes is presented in which the first affects only a constant number of vertices, requiring

the second to process the entire graph even though $\|\delta\|$ is still bounded. This method is used to show that the Single-Source Reachability problem is non-δ-incremental for locally persistent algorithms. The second method is a problem reduction approach similar to that in [Rei87]. This method is used to extend the reachability result to Single-Source (or Single-Sink) Closed-Semiring Path problems and Meet-Semilattice Data-Flow Analysis problems.

5.4 New δ-Analysis Lower Bounds for Undirected Graphs

In this section we apply the techniques discussed above to several problems of undirected graphs. These problems are particularly interesting because they have good incremental update algorithms. The discrepancy between the simultaneous existence of incremental update algorithms and non-δ-incremental lower bounds will be addressed in Chapter 7.

5.4.1 Connectivity

Definition 5.4.1 For a graph G as defined in Section 2.2, we say that two vertices w and x are *connected* in G if and only if there is a path between w and x. A *connected component* is a subset $V' \subset V$ where for every pair $u, v \in V'$, u and v are connected, and for every pair $w \in V'$, $x \in V - V'$, w and x are *not* connected. In other words, a connected component consists of a maximal set of mutually connected vertices. A *connected component labeling* of a graph is a mapping $\ell : V \mapsto \mathbf{N}$ where $\ell(u) = \ell(v)$ if and only if u and v are in the same connected component.

There are various formulations of the connectivity problem in a graph, some of which are not appropriate for the incremental update model. For example, the *connected components* problem is typically phrased: "Given a graph, list its connected components." Since this will always require time proportional to the number of vertices, the incremental version of this problem has a trivial lower bound of n, and hence is unbounded for $\|\delta\|$ in the local persistence model, since adding or deleting some edges has no effect on the output. Another version of the problem is "Maintain a data structure that can answer queries of the form IsConnected(u, v) in constant time as edges (and possibly

vertices) are added and deleted." In the local persistence model, we can formulate this version as a problem of maintaining labelings on the vertices of a graph.

Definition 5.4.2 The *Incremental Connected Component Labeling Problem* is the problem of maintaining a connected component labeling for a graph with addition and deletion of edges, such that the label $\ell(v)$ for each vertex v is stored in the local memory associated with vertex v, and $\ell(u) = \ell(v)$ if and only if u and v are connected.

Theorem 5.4.3 *The Incremental Connected Component Labeling Problem is non-δ-incremental in the local persistence model of computation.*

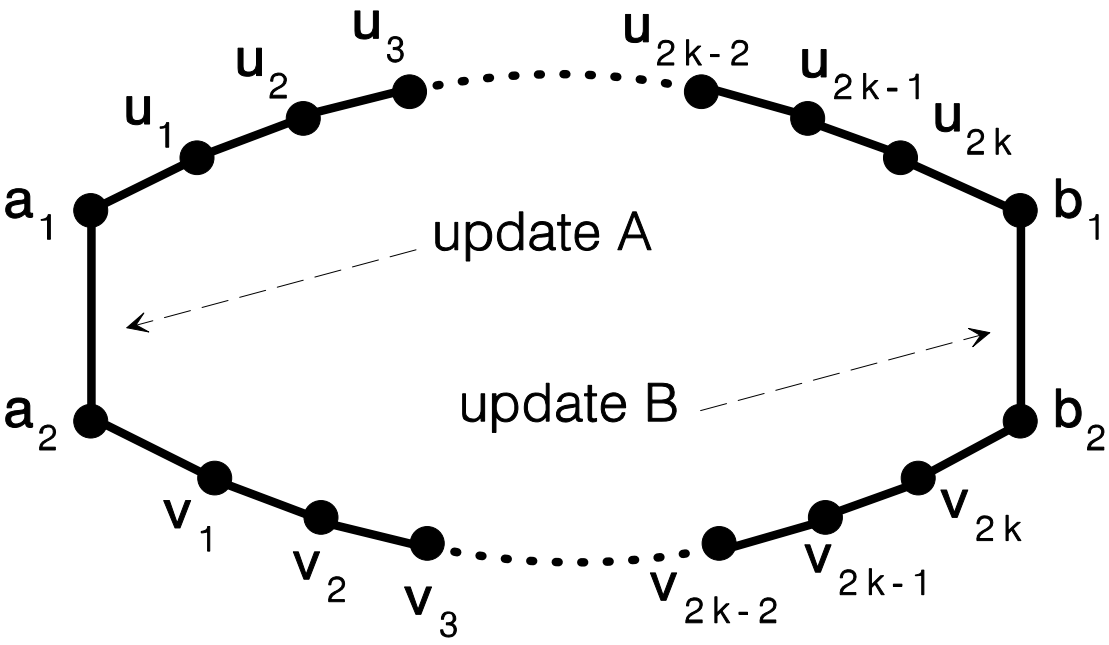

Figure 5.1: Graph for Theorem 5.4.3. Updates A and B are edge deletions. Dashed lines represent chains of vertices.

Proof Consider a graph in the form of a ring, as shown in Figure 5.1. Assume, contrary to the theorem, that there exists a locally persistent algorithm that, in time $f(\|\delta\|)$, can update the graph after an incremental change. We will consider two sequences of changes in the graph, and show that this assumption leads to a contradiction. The idea behind the proof is to show an example in which the algorithm cannot distinguish between a small change and a big change without looking at $O(n)$ vertices; since the algorithm has to do this much work even for the small change, it cannot be δ-incremental.

In the example graph, removing edge a_1a_2 (update A) will leave the graph connected; hence the only work required is to update the neighbor lists for a_1 and a_2, and $\delta = 2$. How much work can a δ-incremental algorithm perform in this case? If such an algorithm exists, the number of vertices it visits must be bounded by a function of δ. Since δ is a constant, the number of vertices visited must also be a constant, i.e., $f(\delta) \leq k$.

Now suppose that update B occurs after update A. The removal of edge b_1b_2 splits the graph into two components each containing $n/2$ vertices; thus the algorithm must visit half the vertices and update them. The problem for the algorithm is determining whether or not update A has previously been made. If it has, the δ-incremental algorithm can have visited only k contiguous vertices. Thus update A can have gone no further than u_{k-2} or v_{k-2}. Thus to distinguish between the two changes the algorithm would have to visit one or the other of u_{k-2} or v_{k-2}, both of which are more than k vertices distant from b_1 and b_2. However, if the algorithm visits these vertices and determines that update A has *not* been made, it will have already visited more than k vertices for a change of size 2. Thus we conclude that no δ-incremental algorithm exists in the local persistence model. ■

Note that Theorem 5.4.3 can be extended to something stronger — any incremental locally persistent algorithm has a lower bound $\Omega(n)$. The proof simply need observe that the incremental algorithm has to look at a constant fraction of the vertices in order to work correctly; in Theorem 5.4.3, the fraction is $> 1/4$. This extension can be applied to Theorems 5.4.5 and 5.5.2 below, and Proposition 4.2 in Ramalingam and Reps [RR91, p. 27].

5.4.2 Biconnectivity

Definition 5.4.4 Two vertices x and y in a graph $G = (V, E)$ are *biconnected* if they are connected by at least two disjoint paths. That is: G has two paths $u_0, u_1, \ldots, u_k$, where $u_0 = x$ and $u_k = y$, and $v_0, v_1, \ldots, v_l$, where $v_0 = x$ and $v_l = y$; and $v_i \neq u_j$ for $0 < i < k$ and $0 < j < l$. The *Biconnected Components Problem* is to partition V into

maximal disjoint sets — the biconnected components — such that all elements of each set are biconnected. The *Incremental Biconnected Components Labeling Problem* is the problem of maintaining a biconnected component labeling for a graph with addition and deletion of edges, such that the label $\ell(v)$ for each vertex v is stored in the local memory associated with v, and $\ell(u) = \ell(v)$ if and only if u and v are biconnected.

Theorem 5.4.5 *The Incremental Biconnected Components Labeling Problem is non-δ-incremental in the local persistence model of computation.*

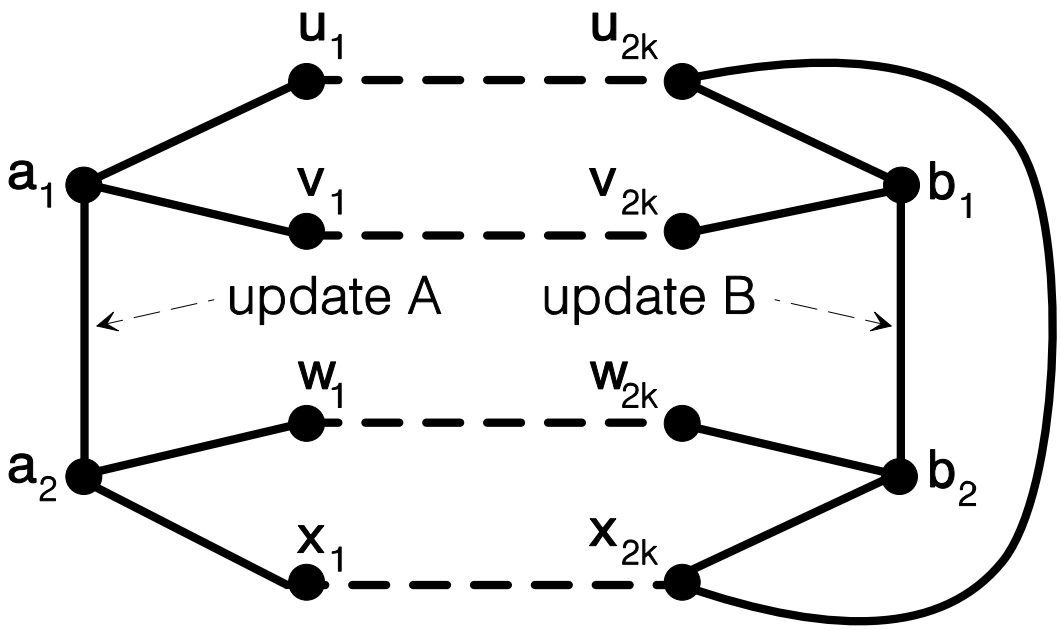

Figure 5.2: Graph for Theorem 5.4.5. Updates A and B are edge deletions. Dashed lines represent chains of vertices.

Proof The argument is quite similar to that in Theorem 5.4.3. Refer to Figure 5.2. Initially, the graph is a single biconnected component, since for any pair of vertices there are at least 2 disjoint paths between them. If edge a_1a_2 is removed (update A), the graph remains biconnected. Thus $|\delta|$ for this change is 2, and the number of vertices visited by any δ-incremental, locally persistent algorithm, $f(|\delta|)$, must be bounded by some constant, $f(2) \leq k$. The vertices visited by the algorithm could include some

of the vertices in the four chains $u_1 \ldots u_{2k}$, $v_1 \ldots v_{2k}$, $w_1 \ldots w_{2k}$, $x_1 \ldots x_{2k}$, but since locally persistent algorithms must visit contiguous vertices, it cannot look farther away than a vertex labeled $k-2$, such as u_{k-2}.

If update B — removal of edge $b_1 b_2$ — follows update A, the second update will split the graph into two biconnected components, since the only path between b_1 and b_2 would be b_1, u_{2k}, x_{2k}, b_2. Thus half, or $O(n)$, of the labels would need to be replaced. If, however, update B occurs before update A, $|\delta|$ is 2, and the vertices visited are bounded by k. The algorithm cannot distinguish between the two possibilities without looking at one of the vertices labeled $k-2$ or smaller — but these vertices are all more than k steps away. Since no bounded incremental algorithm can distinguish between the two sequences, the problem must be non-δ-incremental. ■

5.5 An Extension to the Local Persistence Model

5.5.1 The Extension

In this section we consider an extension to the local persistence model in which information is associated with the *edges* of the graph in addition to, or alternatively to, the vertices. Other than this change, the model remains the same. One could argue that the effect of this change could be made without actually associating information with edges, since each vertex could keep track of the information associated with the edges incident upon it, but for certain problems it is more natural to view the information as being stored with the edges of the graph. We refer to this model as *Edge-Local Persistence*.

5.5.2 Application to Minimum Spanning Tree

Definition 5.5.1 For a connected, undirected graph $G = (V, E)$, a *spanning tree* for G is a graph T where $T \subseteq G$, T connects all vertices in G, and T is acyclic. If G is edge-weighted, i.e., there is a map $w : E \mapsto \Re$, then the *Minimum Spanning Tree* is the

T such that

$$w(T) = \sum_{e \in E} w(e)$$

is minimized. A *Minimum Spanning Tree Edge Labeling* is a map $\ell : E \mapsto \{0, 1\}$ such that

$$\ell(e) = \begin{cases} 1 & \text{if } e \in T \\ 0 & \text{otherwise} \end{cases}$$

The *Incremental Minimum Spanning Tree Edge Labeling Problem* is the problem of maintaining ℓ as edges are added to and deleted from G, and edge weights are modified.

Theorem 5.5.2 *The Incremental Minimum Spanning Tree Edge Labeling Problem is non-δ-incremental in the Edge-Local Persistence model of computation.*

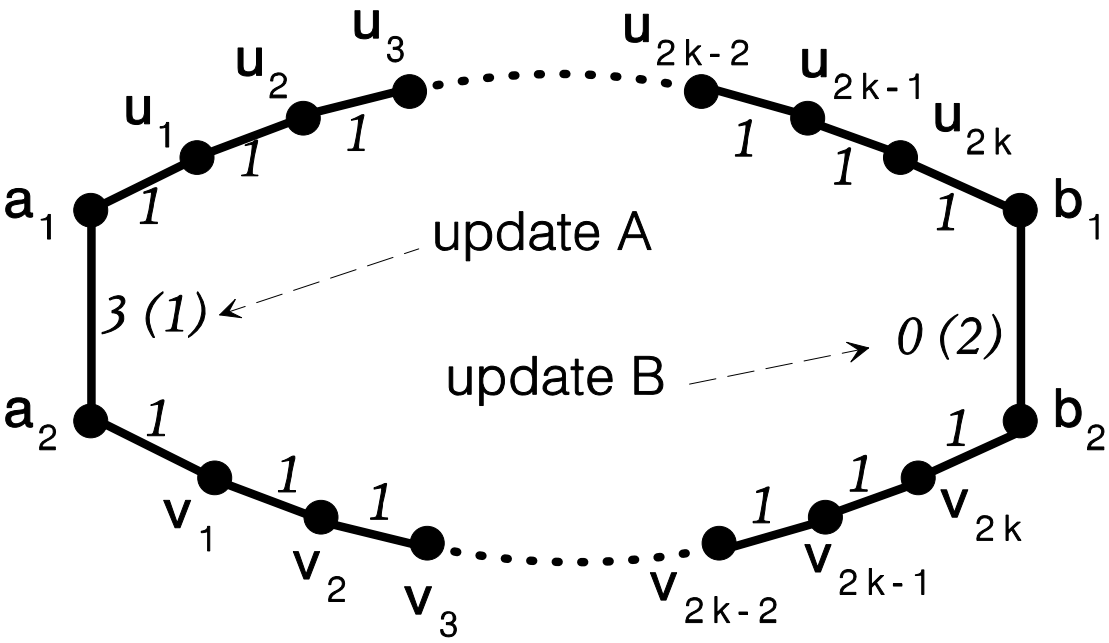

Figure 5.3: Graph for Theorem 5.5.2. Italic numbers are edge weights. Update A is edge weight reduction 3 to 1; update B is edge weight reduction 2 to 0.

Proof Refer to Figure 5.3. The edge weights are the italic numbers in the figure. Initially:

- the weight for edge a_1a_2, $w(a_1a_2)$, is 3;
- the weight for edge b_1b_2, $w(b_1b_2)$, is 0;

- every other edge in the graph has weight 1.

It is easy to see that the initial minimum spanning tree T will include all edges except a_1a_2; hence, the label for a_1a_2, $\ell(a_1a_2)$, is 0, and the label for all other edges, including $\ell(b_1b_2)$, is 1. (Note that the labels are not shown in Figure 5.3.)

Update A is the change $w(a_1a_2) = 3 \Rightarrow w(a_1a_2) = 1$. When this change occurs before update B, there is no change in T, and thus no change in the edge labels — hence $|\delta| = 3$. Therefore, the size of the set of vertices visited by a bounded, locally persistent algorithm for the problem must be bounded by some constant k, and the rightmost vertex that the algorithm can visit is u_{k-2} or v_{k-2}. Update B is the increase in $w(b_1b_2)$ from 0 to 2. When this change occurs subsequent to update A, the label $\ell(b_1b_2)$ goes from 1 to 0. Again, $|\delta|$ is constant, and the size of the set of vertices visited by the algorithm is $\leq k$. The leftmost vertex that the algorithm can visit is u_{k+1} or v_{k+1}.

Suppose instead that when update B is applied to the graph, update A has not taken place. Then the label $\ell(b_1b_2)$ does not change, since b_1b_2 remains in T. There is no common vertex in the graph processed for both updates, and hence there is no way for the algorithm to tell whether or not update A has preceded update B. Hence no such bounded, locally persistent algorithm can exist, and the problem is non-δ-incremental.

■

Chapter 6

Transitive Closure — A Case Study in Incremental Bounds

6.1 The Problem

Given a binary relation $\rightarrow$, if $(a \rightarrow b) \wedge (b \rightarrow c) \Rightarrow (a \rightarrow c)$, $\rightarrow$ is transitive. For a set S, and A a subset of $S \times S$ representing elements related by $\rightarrow$, the closure of A, A^*, is the set such that $(a \rightarrow b) \in A^*$ if and only if there is a sequence $(a_0 \rightarrow a_1), (a_1 \rightarrow a_2), \ldots, (a_{k-1} \rightarrow a_k)$ such that $a_0 = a$ and $a_k = b$, and all $(a_i \rightarrow a_{i+1}) \in A$ for $i = 0$ to $k - 1$. The Transitive Closure problem is to find A^*, given A.

The relationships can be modeled as directed edges in a graph. For example, in intraprocedural data flow analysis, the vertices of the graph represent basic blocks of code, and the edges represent possible paths that program execution can take from one block to another. If, for some vertex x, there exists a path $x, v_1, \ldots, y$, then x *reaches* y. Computing, for each vertex x, the set of vertices that x reaches is called the *reachability* problem; it is easy to see that this is equivalent to computing the Transitive Closure, where each edge is a binary relation.

For the general problem, the Transitive Closure can be computed in the time it takes to perform matrix multiplication, which is $O(n^{2.376})$ [CLR91, CW87]. There is no known lower bound for this problem other than the obvious $\Omega(n^2)$ — the size of the original set of relations is $|S| \times |S| = n^2$. A closely related problem is computing reachability in a directed graph with no cycles, called a *directed acyclic graph* or *DAG*. Transitive closure on a DAG can be computed in $O(n|E_{red}|)$. E_{red} is the set of edges in the *reduced graph* for G, that is, the smallest graph with the same transitive closure as G [Meh84]. In the worst case, $E_{red} = O(n^2)$, so the DAG algorithm requires $O(n^3)$; as a practical matter it is much faster than the standard multiplicative algorithm for

many graphs.

6.2 Incremental Upper Bounds

There has been quite a bit of work in developing incremental algorithms for this problem, beginning with Ibaraki and Katoh [IK83].[1] Ibaraki and Katoh give algorithms for updating the transitive closure for a graph with edge additions and edge deletions. After each change in the graph, the transitive closure information can be accessed in constant time. That is, the cost of a query PATHEXISTS(A,B), which returns "true" if there is a path from a to b in G and "false" otherwise, is $O(1)$. For edge additions, they give a method requiring $O(n^3)$ time to add $q = O(n^2)$ edges. Thus the cost per edge addition is $O(n)$ time, *amortized* across $O(n^2)$ additions; it is possible that a single update could require $O(n^3)$ in the worst case. Typical amortization results consider the cost per operation when $O(n)$ operations are performed; from this point of view the amortized cost is $O(n^2)$. Edge deletions from a graph initially containing m edges require $O(n^2(n+m)) = O(n^4)$ time for $q \leq m$ deletions. This yields an amortized time of $O(n^2)$ per operation for $O(n^2)$ operations, or $O(n^3)$ amortized time for $O(n)$ deletions. In the worst case a single deletion could require $O(n^4)$ time which is substantially worse than starting from scratch. Note that Ibaraki and Katoh's algorithms are partially-dynamic; that is, they handle additions only or deletions only, but not a mix. They can be extended to handle both additions and deletions, but it would appear that in this case the worst case performance could be quite bad, and the amortized performance no better since it would be possible to get in a situation where the worst case was repeated an arbitrary number of times.

In [Ita86] Italiano introduced an algorithm that can process a sequence of n edge additions and SEARCHPATH operations in $O(n)$ amortized time per operation. Hence a sequence of $q = O(m)$ edge additions requires $O(nm)$ time, an improvement over Ibaraki and Katoh's method when $m \ll n^2$. The operation SEARCHPATH(X,Y) returns an arbitrary path from x to y in G if such a path exists, otherwise NULL. Italiano's

[1] This problem was addressed implicitly by Cheston and Corneil in [CC82]; their methods do not yield any improvement over the start-over algorithm in the worst case.

method supports a simple reachability query PATHEXISTS(A,B) in $O(1)$ time. A single addition (or SEARCHPATH) can require $O(nm)$ time in the worst case. Deletions are not addressed.

Italiano has also developed algorithms for updates on DAG's [Ita88]. These algorithms support a sequence of edge deletions in $O(n)$ amortized time per deletion. A sequence of any number of deletions $\leq m$ requires a total of $O(nm)$ in the worst case. This improves on Ibaraki and Katoh's result when $m \gg n$. Again, it is possible that in the worst case a single deletion could require $O(nm)$ time. Similar results were reported independently in [LPv88], who also extend their deletion method to general digraphs, at a higher time complexity.

More recently, Yellin has improved on these when the degree of the graph is bounded [Yel91]. His approach handles a sequence of m insertions in time $O(dm^*)$, where d is the degree bound and m^* is the size of the transitive closure. Thus the cost per operation is $O(\frac{dm^*}{m})$, which, depending on the relative size of the transitive closure, ranges from $O(d)$ to $O(dn)$. Similar results can be achieved for deletions in the case of a DAG. These algorithms will work with a mix of additions and deletions, but without a guarantee that the time bounds will be any better than recomputing each time.

Note that the algorithms described here have the following characteristics:

- All of the results are amortized; none of them has worst-case performance guaranteed to be any better than recomputing.
- No method is guaranteed to do better than recomputing, even in an amortized sense, for the fully-dynamic problem, i.e., for a mix of additions and deletions.
- None of them has been analyzed from the δ point of view, so it is not clear that they will work quickly for "small" updates.
- Deletions, particularly for graphs containing cycles, appear to be more difficult for this problem than insertions.

In the subsequent sections we consider lower bounds for transitive closure, and show how they relate to the upper bounds in this section.

6.3 IRLB Analysis

We now derive lower bounds for incremental update of transitive closure problems, using IRLB's.

6.3.1 Arbitrary Transitive Relations

Theorem 6.3.1 *Transitive Closure (TC) has a* **Class 3** *IRLB; that is, given that the IRLB assumptions hold, fully-dynamic incremental algorithms can do no better, in the worst case, than starting over.*

Proof Refer to the IRLB proof procedure in Figure 4.1 on page 17 and the basic theorem for IRLB's (Theorem 4.2.4 on page 20). To build a fast initialization for transitive closure, add new elements ξ and ϕ to the input set, plus the relation $\xi \rightarrow \phi$. For each element a in the original input set, we add two relations: $a \rightarrow \xi$ and $\phi \rightarrow a$. This is accomplished in steps 1–3, in time proportional to the length of the input. With these modifications, $x \rightarrow y$ for any pair (x, y) in the original input, since $x \rightarrow \xi \rightarrow \phi \rightarrow y$. Thus in the initial solution, computed in step 4, A^* contains all possible pairs. Then in a single iteration of the loop in steps 6–9, the relation $\xi \rightarrow \phi$ is removed. Now it is easy to see that the original elements in the resulting relation have the same transitive closure as they did in the original input set. Here $\epsilon(\ell)$ is 1; hence the transitive closure function has an IRLB of 1, and is in **Class 3**. ■

The proof is illustrated in Figures 6.1 and 6.2. Figure 6.1 represents an example graph; Figure 6.2 illustrates the graph after the transformation described by the proof has been applied. The dotted and dashed lines represent the relations added in steps 1–3; the dashed line corresponds to the relation that will be removed in step 7.

6.3.2 Directed Acyclic Graphs

The proof construction for Theorem 6.3.1 adds cycles to the graph. For example, see $A \rightarrow \xi \rightarrow \phi \rightarrow A$ in Figure 6.2. Hence this Theorem does not apply to DAG's. Instead,

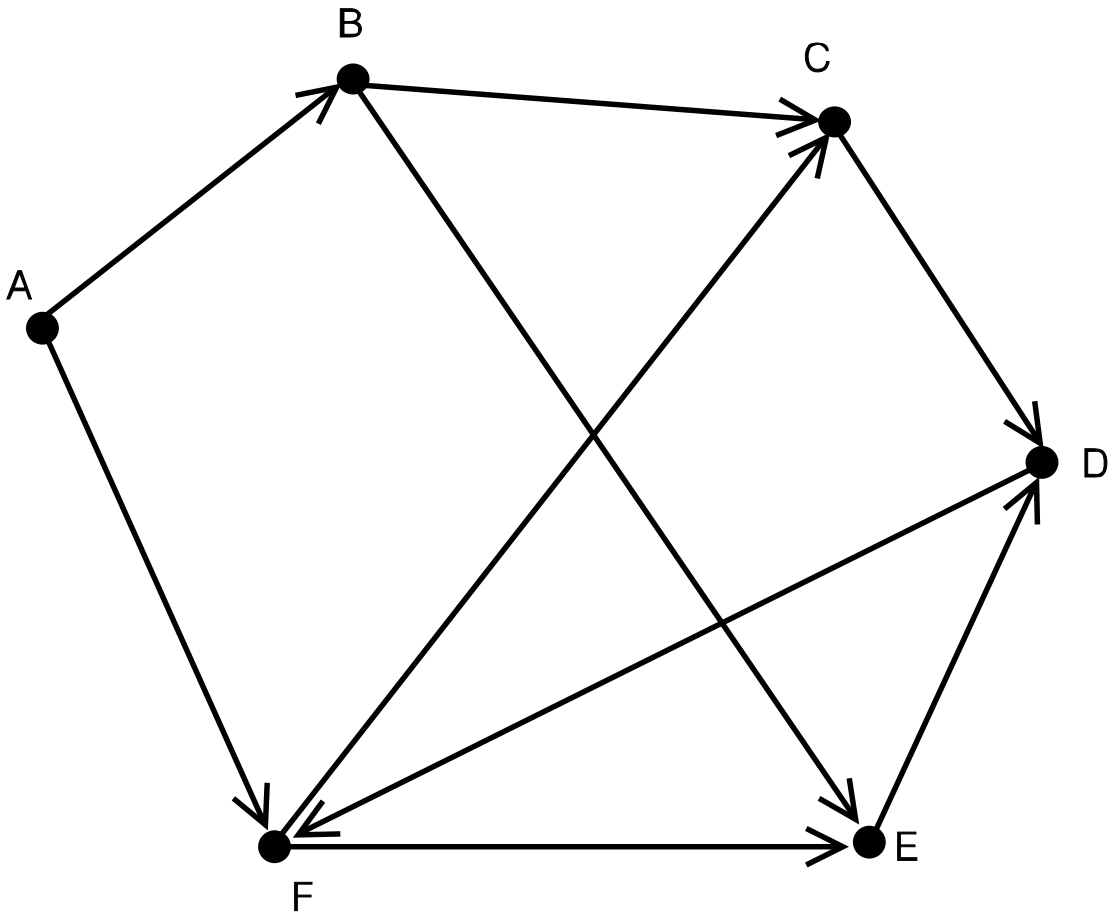

Figure 6.1: Example graph for transitive closure.

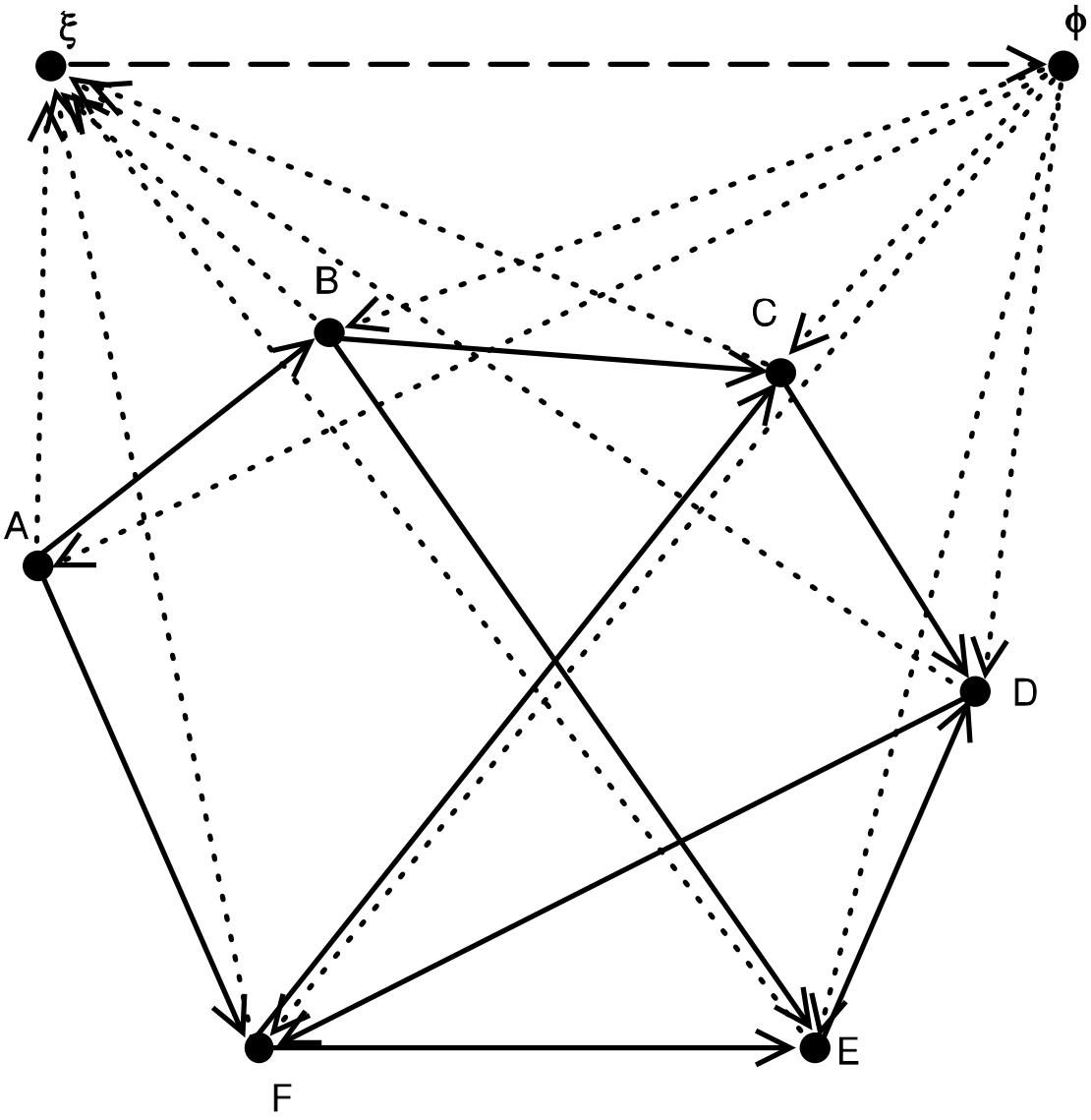

Figure 6.2: Example graph with transformations for IRLB proof procedure. Dotted and dashed lines represent relations added in steps 1–3; dashed line will be removed in step 7.

we can show that Transitive Closure on DAG's (TC-DAG) has a Class 2 IRLB, given the following conjecture.

Conjecture 6.3.2 *TC-DAG requires time $\omega(n+m)$ to compute.*

As mentioned previously, it is clear that TC is $\Omega(n^2)$ since the size of the answer can be as large as n^2.

Theorem 6.3.3 *TC-DAG has a Class 2 IRLB; that is, given that the IRLB assumptions hold, an incremental algorithm for TC-DAG will require $\frac{1}{n}$ of the time of the optimal start-over algorithm in the worst case.*[2]

Proof The proof for TC-DAG requires a more subtle fast initialization than most of the others seen in this work. Referring to the IRLB proof procedure (Figure 4.1), we will add a step 0.5 to compute a topological sort of the input vertices. A topological sort for a directed graph $G = (V, E)$ is an onto mapping $\tau : V \mapsto \{1 \ldots n\}$ such that for every edge $xy \in E$, $\tau(x) < \tau(y)$. It is well-known that τ exists if and only if G is acyclic; the mapping can be found in linear time ($O(n+m)$) [Meh84].

In conjunction with the computation of the topological sort, we will maintain an inverted list of the vertices so that we can traverse them in topologically sorted order. Let $x = \tau^{-1}(i)$ be the vertex such that $\tau(x) = i$. For the loop in steps 1–3, $\delta(\ell) = n-1$.[3] In step 2, if there is no edge from $\tau^{-1}(i)$ to $\tau^{-1}(i+1)$, add it, and make a note of the addition in a list L.

We can now, in step 4, quickly solve the initial problem. Observe for any vertex x where $\tau(x) = i$, x cannot reach any of the vertices with labels $1 \ldots i-1$; this follows easily from the definition of τ. Furthermore, for any vertex y such that $\tau(y) > \tau(x)$, x reaches y, since either the path $x, \tau^{-1}(i+1), \ldots, \tau^{-1}(\tau(y)-1), y$ existed in the original graph, or else we added any missing edges in steps 1–3. Therefore, vertex x reaches vertices $i, \ldots, n$.

---

[2] n here refers to the number of vertices in the graph; the size of the input $l = n + m$, i.e., vertices plus edges.

[3] Note that the symbol δ is overloaded here. In the context of an IRLB proof, δ is the function in the IRLB Procedure in Figure 4.1, indicating the amount of preprocessing required as a function of the input size.

To complete the proof procedure, the loop in steps 6–9 will be iterated once for each item in list L. The incremental changes consist of removing each of the edges that was added in step 2.

In order for Theorem 4.2.4 to apply, it must be shown that the additional time required by the procedure described here is dominated by the cost of producing a transitive closure. The cost of producing the topological sort is $O(n+m)$. Assuming step 2 can be performed in constant time,[4] the loop in steps 1–3 requires $O(n)$ time. Step 7 is clearly constant time, and will be performed $O(n)$ times. Hence the total "extra" work for the procedure is $O(n+m)$; given 6.3.2, the time to compute TC-DAG from scratch dominates $O(n+m)$, and hence we can apply Theorem 4.2.4. Since $\delta(\ell) = n$, TC-DAG is classified as a Class 2 problem. ■

The transformation used in the proof is illustrated in Figures 6.3 and 6.4.

6.4 δ-incremental Algorithms for Reachability Problems on DAG's

In the following we develop an incremental algorithm that is efficient, in the δ-analysis sense, for updating Transitive Closure on a Directed Acyclic Graph (TC-DAG).[5] We first develop algorithms for updating the closely related problem of Single-Sink Reachability on a DAG (SSR-DAG). Initially, we will assume that no edges are added that induce cycles in the graph; later we will consider the results of relaxing that restriction.

6.4.1 Relationship to All Pairs Shortest-Path Problem

Ramalingam and Reps give a δ-bounded incremental algorithm for solving the all pairs shortest-path problem for graphs with positive edge weights (APSP>0) [RR91, pp.

[4]There is a data structures issue here; in order for the step to be performed in constant time, there must be a way to determine whether there is an edge between two arbitrary vertices in constant time. If the graph is presented as an adjacency matrix, clearly there is no problem. If instead the graph is presented as a linked list or in some other form, then an adjacency matrix can be constructed in linear time as the graph is read in, using an implementation that supports constant time initialization; see, e.g., [LD91, pp. 136–138].

[5]We understand that Ramalingam and Reps may have independently developed algorithms for the closely related problem of updating reachability on reducible flow graphs [Rep92].

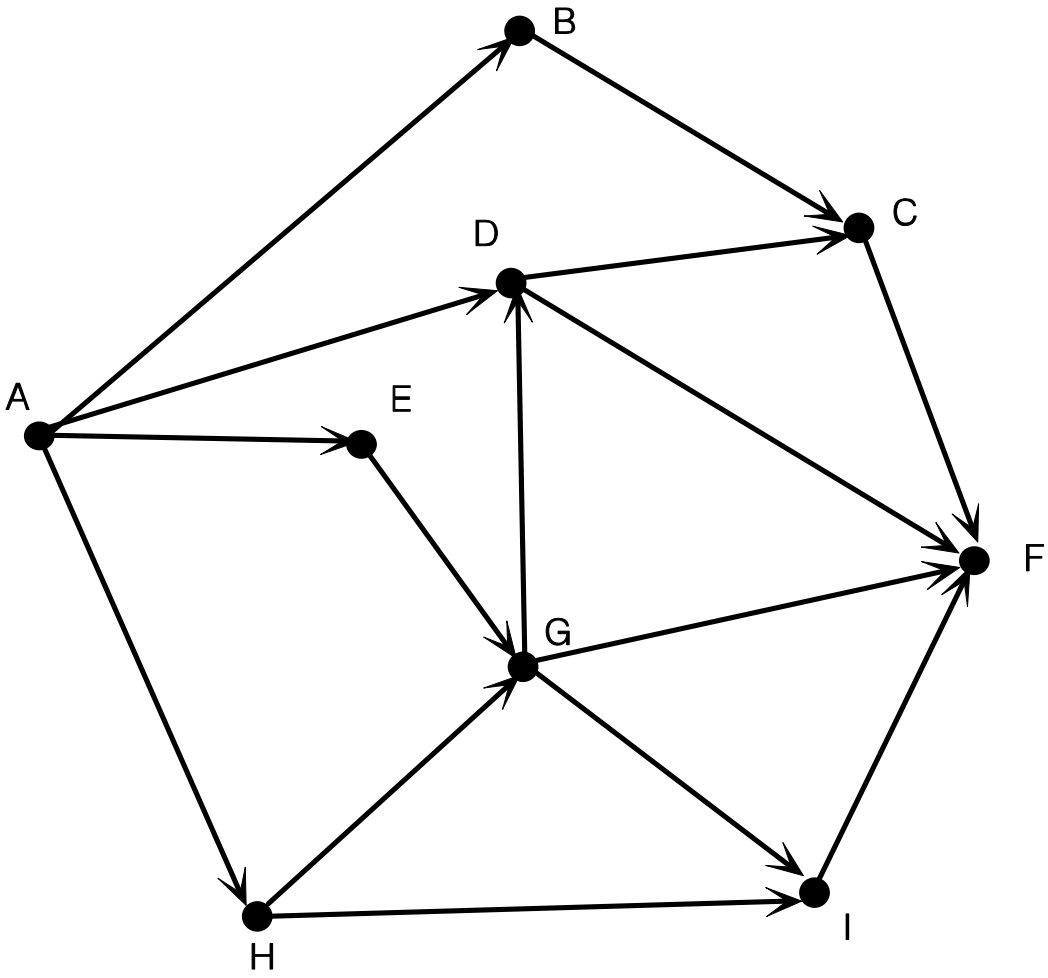

Figure 6.3: Example graph for transitive closure on a DAG.

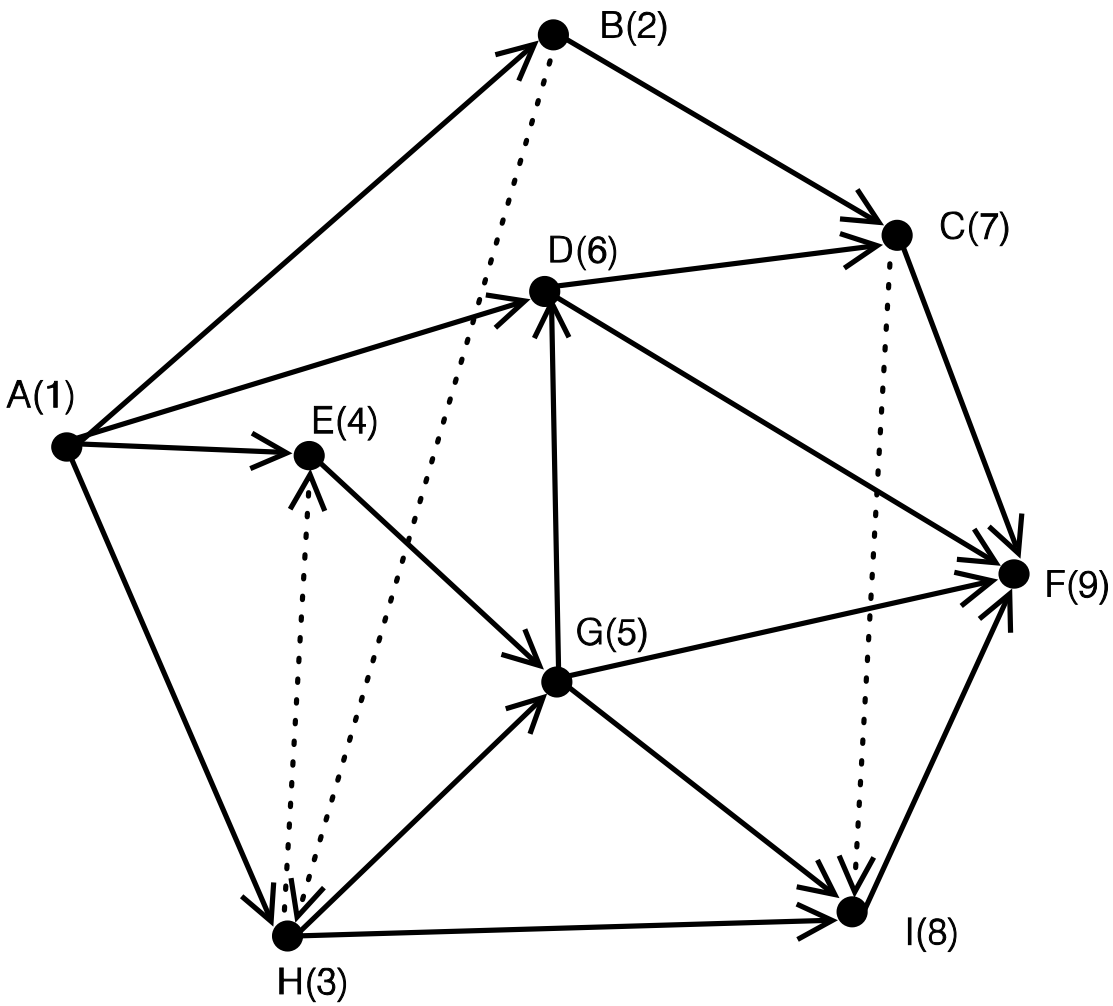

Figure 6.4: Example graph with transformations for IRLB proof procedure. The labels in parentheses represent the topological sort mapping. Dashed lines represent edges added in steps 2 and removed in step 7.

14–20]. At first glance, it would appear that this would yield a bounded algorithm for transitive closure:

1. set all edge weights to 1;
2. run the incremental APSP>0 algorithm;
3. if the weight of a path from x to y is $< \infty$, then x reaches y, otherwise x does not reach y.

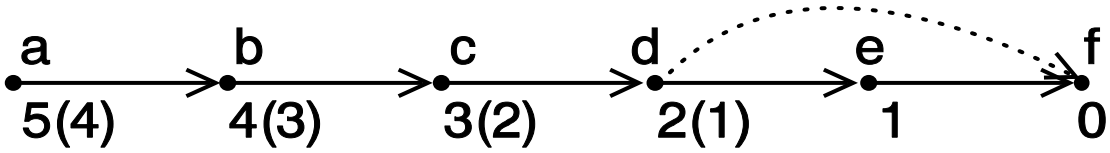

Figure 6.5: Illustration of fallacy in calling APSP>0 algorithm a bounded algorithm for incremental transitive closure. Dotted line represents incremental addition.

However, to draw the conclusion that this is a bounded algorithm in δ analysis would be fallacious. During the operation of this algorithm, path weights may change at various vertices, without changing reachability. For an example of this, consider Figure 6.5. When the edge df is added, the weights at all vertices from a to d must be updated. Clearly, a chain like this can be $O(n)$ in length. But from the perspective of transitive closure, $|\delta| = O(1)$ since the only change required is to note the new edge at vertices d and f — there is no change in reachability for any vertex. Hence the work performed by the incremental algorithm will be unbounded for this change. Thus we see that an APSP>0 algorithm will not yield a bounded algorithm for transitive closure.

6.4.2 Directed Graphs with Cycles

The transitive closure problem determines, for each pair of vertices x and y, whether there is a path from x to y. A closely related problem is to determine, for each vertex x and a distinguished vertex s, whether there is a path from s to x. Vertex s is referred to as the *source*. This is called the *Single-Source Reachability* problem or *SS-REACH*. There is also the dual problem of determining, for each vertex x and a distinguished vertex k, whether there is a path from x to k; k is called the *sink* and the problem the *Single-Sink Reachability* problem.

In [RR91] it is shown that SS-REACH is non-δ-incremental in the local persistence model (see §5.3.1). This is demonstrated with the graph reproduced in Figure 6.6. The

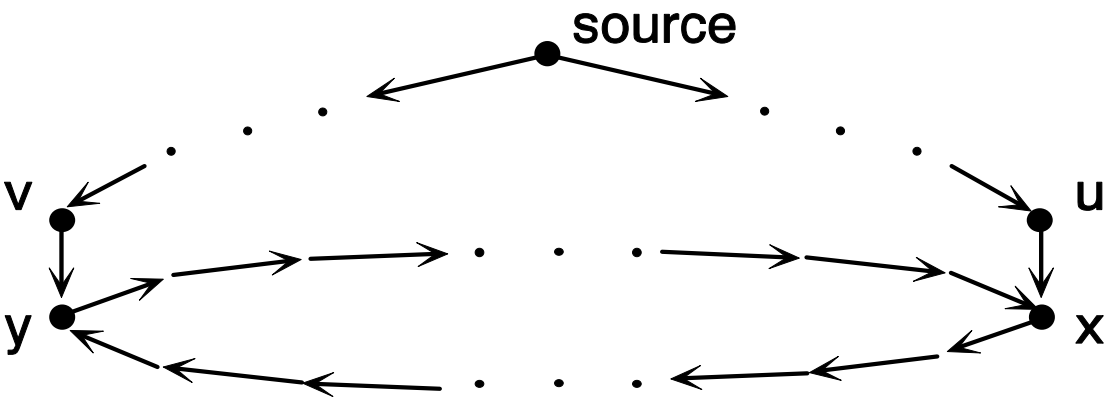

Figure 6.6: Demonstration that SS-REACH is non-δ-incremental. Dots represent arbitrarily long chains of edges.

argument is based on the observation that when edge vy is deleted, there is no way to tell if ux is present without traversing more than a constant number of vertices.

The proof of the non-δ-incrementality of Single-Source Reachability can easily be reformulated to prove the same fact for Single-Sink Reachability. Note that this argument depends critically on the presence of a cycle in the graph and thus has no implication for DAG's. Note also that the non-δ-incrementality of SS-REACH does *not* imply that Transitive Closure is non-δ-incremental, since changes that might be local for SS-REACH could be non-local for the general problem. Consider what happens in Figure 6.6 when edge vy is deleted. Regardless of the presence or absence of edge ux, any update procedure will have to traverse the graph from v all the way back to the source, updating the vertices along the way to indicate that they can no longer reach the vertices in the cycle at the bottom of the figure.

In the sections below we develop δ-incremental algorithms for Single-Sink Reachability on a DAG and Transitive Closure on a DAG. The existence of a Transitive Closure δ-incremental algorithm for an arbitrary digraph remains open.

6.5 Single-Sink Reachability on a DAG

6.5.1 Preconditions

In this section we present a δ-incremental algorithm for Single-Sink Reachability on a DAG, which we denote SSR-DAG. The algorithm can be reformulated to yield a δ-incremental algorithm for the dual problem, Single-Source Reachability on a DAG, with the same complexity.

We assume that the state of the problem, prior to any incremental changes, is as follows:

- a list of predecessors, PRED(x), i.e., all vertices y such that $yx \in E$, is associated with each vertex x;
- a list of successors, SUCC(x), i.e., all vertices y such that $xy \in E$, is associated with each vertex x;
- each vertex x has a Boolean variable, REACHES, which is **true** if and only if there is a path from x to the sink vertex k;
- the graph is acyclic.

6.5.2 Edge Additions for SSR-DAG

When an edge is added from x to y, it may affect the value of REACHES for x, and potentially for *ancestors* of x. A vertex z is an ancestor of x, written $z \rightsquigarrow x$, if there is a path from z to x. The addition procedure is shown in Figure 6.7 and described below. This procedure *assumes* that after the addition of edge xy, the graph remains a DAG; i.e., xy does not induce a cycle.[6]

In this procedure, and for the procedures to follow, we assume that we have a queue package available, with the following characteristics:

- **emptyQueue**(q): initializes q to be an empty queue.

[6]By using the priority ordering algorithm in [AHR+90], cycle-inducing edges can be detected in time $O(\|\delta^2\| \log \|\delta\|)$, but of course in this case the algorithm would not be δ-incremental, by a similar argument to that in §6.4.1.

- **enqueue**(q, x): adds element x to the end of queue q.
- **dequeue**(q, x): removes an element from the front of queue q and returns it in x.
- **isEmpty?**(q): predicate, *true* if q is an empty queue and *false* otherwise.

The elements enqueued and dequeued can be any arbitrary data type. The notation (x, y) represents the ordered pair x and y; and if $p = (x, y)$, then $p[1] = x$ and $p[2] = y$. The notation REACHES(x) is assumed to return the value of the REACHES flag for vertex x.

```
    AddEdgeSSR-DAG(x,y)
1.      emptyQueue(q)
2.      if not REACHES(x) & REACHES(y) then
3.          enqueue(q, x)
4.          REACHES(x) ← true
5.      while not isEmpty?(q) do
6.          dequeue(q, z)
7.          for each v ∈ PRED(z) do
8.              if not REACHES(v) then
9.                  enqueue(q, v)
10.                 REACHES(z) ← true
```

Figure 6.7: Procedure AddEdgeSSR-DAG

AddEdgeSSR-DAG works by first determining whether or not the addition of xy changes the state of REACHES(x) — if not, we are done. If so, REACHES$(x) \leftarrow$ **true**, and x is placed in a queue q. Thereafter, each time a vertex is dequeued, each of its predecessors is checked for an update, and if the value of REACHES goes from **false** to **true** the predecessor is placed in the queue. Processing terminates when the queue is empty. We refer to a vertex x as *active* if the update causes the value of REACHES(x) to change.

Lemma 6.5.1 *If vertex v is active as the result of an incremental edge addition xy, then one of the following must be true:*

1. $v = x$;

2. *for some* $z \in \text{SUCC}(v)$, *the value of* $\text{REACHES}(z)$ *changes as the result of the update, i.e., z is active.*

Proof Suppose that neither condition holds for v. Before the incremental change,

$$\text{REACHES}(v) = \bigvee_{z=\text{SUCC}(v)} \text{REACHES}(z)$$

held for v. Since $\text{SUCC}(v)$ has not changed, and none of the REACHES flags for successors of v has changed, the equation still holds, and hence v cannot be active. ■

Lemma 6.5.2 *A vertex v can be active after edge xy is added (or deleted) only if* $v \rightsquigarrow x$.

Proof If there is no path from v to x, then the addition of edge xy cannot possibly result in v reaching the sink afterwards unless it already did before the change. Likewise, in the absence of a path from v to x, the deletion of edge xy cannot eliminate a path from v to the sink. Hence v is not active. ■

Theorem 6.5.3 *Procedure AddEdgeSSR-DAG in Figure 6.7 correctly updates* REACHES *for all vertices when an edge is added to a DAG, assuming that the added edge does not create a cycle.*

Proof From Lemma 6.5.1, we know that the only vertices that need to be updated are the base of the added edge (x), and vertices whose immediate successors have changes in REACHES. Clearly $\text{REACHES}(x)$ is updated correctly by line 2 of 6.7. When x is examined on the first pass through the main loop at line 5, each predecessor of x will be placed in the queue if and only if it can reach the sink only as a result of the update. Inductively, since each time a vertex is dequeued its immediate predecessors will be considered, we see that if a vertex has an immediate successor whose REACHES flag changes, then it will be enqueued and tested. Hence the vertices identified by Lemma 6.5.1 will be considered by the procedure, and updated if necessary. ■

6.5.3 Analysis of Procedure AddEdgeSSR-DAG

For this problem, $\|\delta\|$ consists of the neighborhood of the subset of vertices that are active. When a vertex is enqueued in line 9, its REACHES flag is set to **true**, and thus the test in line 8 guards against it ever being enqueued again. Hence each active vertex is enqueued exactly once. Each time a vertex is dequeued, the work performed by the procedure is to examine each predecessor. This proves the following theorem.

Theorem 6.5.4 *Procedure AddEdgeSSR-DAG in Figure 6.7 requires time* $O(\|\delta\|)$.

6.5.4 Edge Deletions for SSR-DAG

Edge deletion appears to be a more difficult change to update than edge addition. Consider the situation when updating REACHES for a particular vertex, say u. Suppose that the addition of an edge has changed the value of REACHES for some successor v of u, and REACHES(u) = **false**. Then we can simply set REACHES$(u) \leftarrow$ **true**. Now suppose instead that a v can no longer reach z as the result of a deletion, and REACHES(u) = **true**. It would be a mistake to simply set REACH$(u) \leftarrow$ **false**. The problem is that there may be another path from u to k that does not pass through v. Thus before setting REACHES(u) to **false** it must be determined that no such alternate path exists. Furthermore, this determination can only be made *after* all active successors of u are updated. To see why, consider Figure 6.8. When edge xy is deleted, x no longer has a path to k and hence REACHES(x) = **false**. When vertex a is checked as a predecessor of x, REACHES(b) will still be **true**, and thus REACHES(a) will not be updated correctly.

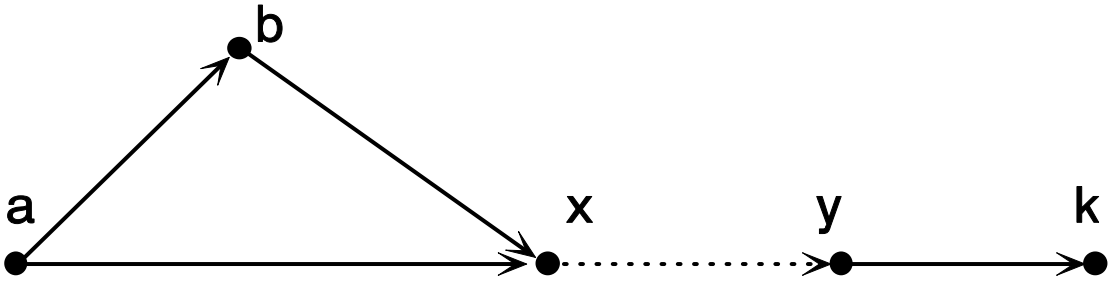

Figure 6.8: Illustration of difficulty in updating after deletion.

For the SSR-DAG problem, this is not too hard to handle, as long as we are willing

the visit the same vertex more than once. If REACHES(u) is updated before one of its successors and hence remains **true** even though it ultimately will be set to **false**, that is not really a problem if it gets updated correctly later. Refer again to Figure 6.8. When a is updated as a predecessor of x, it will remain **true**; however, when x is considered, b becomes active and will be added to the queue. At a later step, a will again be considered, this time as a predecessor of b, and since REACHES(b) and REACHES(x) are now up-to-date, REACHES(a) will be set to **false**, as required.

We begin by defining a subroutine, CanReach(u, v), which returns **true** if u can still reach k *after* edge uv is removed, else returns **false**.

```
     CanReach(u,v)
1.        for each z ∈ SUCC(u) − {v}  do
2.            if REACHES(z)  then
3.                return true
4.        return false
```

Figure 6.9: Procedure CanReach

Lemma 6.5.5 *Procedure CanReach*(u, v) *in Figure 6.9 correctly indicates whether vertex* u *could reach the sink* k *without using edge* uv*, assuming that the* REACHES *flags for all successors of* u *are up-to-date.*

Proof If u can still reach k without using edge uv, it must be via a path $u, z, \ldots, k$, where $z \in \text{SUCC}(u) - \{v\}$. Since CanReach checks each candidate successor and returns **true** if and only if there is a successor that can reach k, it works correctly. ■

Lemma 6.5.6 *Procedure CanReach*(u, v) *requires time at most* $O(\text{SUCC}(u))$.

Proof CanReach has a single loop that is iterated once for each vertex in $\text{SUCC}(u) - \{v\}$; the operation in line 2 is $O(1)$. ■

Theorem 6.5.7 *Procedure DeleteEdgeSSR-DAG in Figure 6.10 correctly updates the* REACHES *flags in a DAG after the deletion of an edge* xy.

```
    DeleteEdgeSSR-DAG(x, y)
1.      emptyQueue(q)
2.      if not CanReach(x, y) then
3.          enqueue(q, x)
4.          REACHES(x) ← false
5.      while not isEmpty?(q) do
6.          dequeue(q, z)
7.          for each v ∈ PRED(z) do
8.              if REACHES(v) then
9.                  if not CanReach(v, z) then
10.                     enqueue(q, v)
11.                     REACHES(v) ← false
```

Figure 6.10: Procedure DeleteEdgeSSR-DAG

Proof By the stated preconditions for the procedure, we know that the values of REACHES are correct before we process the update. The only changes we make to REACHES are in lines 4 and 11, in which REACHES is set to **false**. By the logic test in lines 2 and 9, and Lemma 6.5.5, it is clear that a REACHES flag is changed only when a vertex can no longer reach the sink, and thus is updated correctly. To see that all such vertices do get updated, note that Lemma 6.5.1 shows that the only vertices that need to be updated, other than x which is checked by line 2, are those with active successors; furthermore, all active vertices are ancestors of x, by Lemma 6.5.2. Since each active vertex v gets added to q, and then each predecessor of v gets checked when v is dequeued, it is certain that a vertex will be checked if it is a candidate to be active. While its possible for a vertex to be checked more than once, as discussed above, the *last* time that a vertex is a predecessor of an active vertex, all its successors will be up-to-date, and hence it will be updated correctly. ■

Theorem 6.5.8 *Procedure DeleteEdgeSSR-DAG in Figure 6.10 requires time* $O(\|\delta\|_2)$.

Proof An active vertex will be enqueued exactly once — after v is enqueued, REACHES(v) ← **false** thus assuring that it can never be enqueued again because of the test in line 8. Each time a vertex v is visited, the algorithm in the worst case looks

at all successors (in the call to CanReach) of all predecessors (line 7) of v. The successor of a predecessor of v is at most two edges away from v, an active vertex. Hence the total work is bounded by the second order extended size of the set of active vertices, or $O(\|\delta\|_2)$. ■

6.6 Transitive Closure on a DAG

The methods developed above for SSR-DAG are extended to solve the problem of updating Transitive Closure on a DAG, referred to as TC-DAG.

6.6.1 Preconditions

The preconditions for TC-DAG are nearly the same as those for SSR-DAG as presented in §6.5.1. However, since the TC-DAG problem requires maintenance of reachability information for all pairs of vertices, each vertex u will have associated with it a set REACH(u), such that $\forall v \in V$, $v \in$ REACH(u) if and only if $u \rightsquigarrow v$.

6.6.2 Edge Additions for TC-DAG

The operation of AddEdgeTC-DAG can be viewed as consisting of two phases. The first phase works forward in the graph from the point of the change, identifying vertices that have to be added to the REACH set for vertex x, the base of the added edge. When phase one is complete, the set of such vertices will be in the set ADDS. If x could already reach y, then there is no update needed and the rest of the procedure is skipped — ADDS $= \emptyset$. Otherwise, since x did not reach y previously, it is possible that it could not reach some or all of the immediate successors of y. Each successor will be added to the queue q; when a successor vertex is dequeued, REACH(x) will be tested to see if x could reach it, and if not, *its* successors are enqueued, and so forth. When the queue is empty, all candidates to be added to REACH(x) have been checked, and ADDS contains the appropriate vertices, as is shown by Lemma 6.6.2.

```
    AddEdgeTC-DAG(x,y)
        /* phase one */
1.      emptyQueue(q)
2.      if y ∉ REACH(x) then
3.          ADDS ← {y}
4.          enqueue(q, y)
5.          while not isEmpty?(q) do
6.              dequeue(q, z)
7.              for each v ∈ SUCC(z) do
8.                  if v ∉ ADDS & v ∉ REACH(x) then
9.                      ADDS ← ADDS ∪ {v}
10.                     enqueue(q, v)
11.         VISITED ← ∅
12.         enqueue(q, (x, ADDS))
        /* phase two */
13.     while not isEmpty?(q) do
14.         dequeue(q, p)
15.         z ← p[1]
16.         ADDS ← p[2]
17.         for each n ∈ ADDS do
18.             REACH(z) ← REACH(z) ∪ n
19.         for each v ∈ PRED(z) do
20.             if v ∉ VISITED then
21.                 VISITED ← VISITED ∪ {v}
22.                 ADDS ← ∅
23.                 for each n ∈ ADDS do
24.                     if n ∉ REACH(v) then
25.                         ADDS ← ADDS ∪ {n}
26.                 if ADDS ≠ ∅ then
27.                     enqueue(q, (v, ADDS))
```

Figure 6.11: Procedure AddEdgeTC-DAG

Definition 6.6.1 Let $\textsc{diff}(v)$ be the change in vertex v required by an update; i.e., if $\textsc{reach}(v)$ is the reach set before an incremental change, and $\textsc{reach}'(v)$ is the reach set after the change, then

$$\textsc{diff}(v) = (\textsc{reach}(v) \cup \textsc{reach}'(v)) - (\textsc{reach}(v) \cap \textsc{reach}'(v))$$

Let any vertex v, such that $\textsc{diff}(v) \neq \emptyset$, be called an *active vertex*; by extension we can refer to *active successors* or *active predecessors*.

Lemma 6.6.2 *When phase one of Procedure AddEdgeTC-DAG in Figure 6.11 has completed,* $v \in \textsc{adds}$ *if and only if* $v \in \textsc{diff}(x)$.

Proof For a vertex v to be in $\textsc{diff}(x)$, it must satisfy two conditions:

1. it must *not* have been in $\textsc{reach}(x)$ before xy was added;
2. there must be a path $x \rightsquigarrow v$ after xy is added.

To prove the lemma, we must show that exactly the vertices that meet these conditions are in ADDS at the end of phase one. We consider the conditions separately.

The first condition follows easily from the **if** tests in lines 2 and 8 — a vertex can only be added to ADDS if it is not in $\textsc{reach}(x)$.

We demonstrate the second condition by induction. The basis is y — clearly there is a path from x to y after xy is added, and y will be added, in line 3, to ADDS, as long as the first condition was met. For an arbitrary vertex $v \in \textsc{diff}(x)$, observe that if there is a path from x to v after xy is added, but not before, then the path must include edge xy; so there is a path from y to v. Furthermore, every vertex on that path must be in $\textsc{diff}(x)$: if there is some vertex u on the path $y \ldots u \ldots v$, then the only reason that u is not in $\textsc{diff}(x)$ is that $u \in \textsc{reach}(x)$; but then there is a path $x \ldots u \ldots v$ that *does not* include xy, implying $v \in \textsc{reach}(x)$, a contradiction. Thus it is sufficient to show that if $v \in \textsc{diff}(x)$ (the inductive hypothesis), then $z \in \textsc{pred}(v)$ and $z \in \textsc{adds}$ implies $v \in \textsc{adds}$. This is true, because the loop in lines 7–10 checks every successor of a vertex that has been added to ADDS and enqueues it if it is not in ADDS (because in that case it has already been added) and not in $\textsc{reach}(x)$ (condition one). ■

Phase two of the AddEdgeTC-DAG procedure starts at x, the base of the added edge, and works backwards in the graph adding elements to the reach sets of vertices. The vertices added to the reach set for some vertex x are all candidates to be added to the reach sets for the predecessors of x. If no vertices are added to $\text{REACH}(x)$, then none need be added to REACH for the predecessors of x, as shown by Lemma 6.6.4. Furthermore, it is never necessary to visit a vertex more than once, per Lemma 6.6.5. The following Lemma is analogous to Lemma 6.5.1.

Lemma 6.6.3 *If vertex v is active as the result of an incremental edge addition xy, then one of the following must be true:*

1. $v = x$;
2. *for some $z \in \text{SUCC}(v)$, $\text{DIFF}(z) \neq \emptyset$, i.e., z is active.*

Proof Suppose that neither condition holds for v. Before the incremental change,

$$\text{REACH}(v) = \bigcup_{z=\text{SUCC}(v)} \text{REACH}(z)$$

held for v. Since $\text{SUCC}(v)$ has not changed, and none of the REACH sets for successors of v has changed, the equation still holds, and hence $\text{DIFF}(v) = \emptyset$. ■

Note that the proof for Lemma 6.5.2, which states that a vertex v can only be active if $v \rightsquigarrow x$, still holds with the new definition of *active* in Definition 6.6.1.

Lemma 6.6.4 *After an edge xy is inserted, for any $v \neq x$, $\text{DIFF}(v)$ will be a (not necessarily proper) subset of the* DIFF *sets of each of its* active *successors.*

Proof Suppose there is some vertex t such that $t \in \text{DIFF}(v)$ and $t \notin \text{DIFF}(w)$, where $w \in \text{SUCC}(v)$ and w is active. Note that the only way vertex t can be in set $\text{DIFF}(v)$ is if $t \in \text{DIFF}(x)$, where x is the base of the edge added by AddEdgeTC-DAG, and there is a path $v \rightsquigarrow x$. Since w is active, $w \rightsquigarrow x$, by Lemma 6.5.2, so the addition of edge xy created a path $w \ldots xy \ldots t$; thus w can now reach t, so the only reason

$t \notin \text{DIFF}(w)$ is that $t \in \text{REACH}(w)$ *before* the incremental change. But $t \in \text{REACH}(w) \Rightarrow t \in \text{REACH}(v) \Rightarrow t \notin \text{DIFF}(v)$. ■

Lemma 6.6.5 *It is never necessary to visit a vertex more than once. That is, if some vertex z is visited once and then is checked in step 11 of Procedure AddEdgeTC-DAG in Figure 6.11, putting z back in the queue would not change the final value of* $\text{REACH}(z)$.

Proof Consider a vertex z that would be added to q twice in step 13 except for the intervention of the test in step 12. There must be two paths from z to x; let w_1 and w_2 be the two successor vertices on these paths. That is, there are paths $zw_1 \ldots x$ and $zw_2 \ldots x$, not necessarily distinct. This is illustrated in Figure 6.12. Assume, without

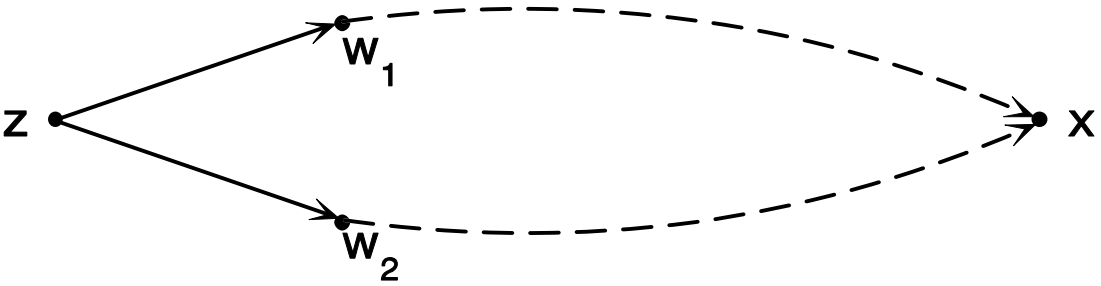

Figure 6.12: Illustration of the conditions for Lemma 6.6.5.

loss of generality, that w_1 is visited first, and z is added to the queue at that time, and that the updated value of $\text{REACH}(z)$ is computed before $\text{REACH}(w_2)$ has been updated. When the later visit to w_2 occurs, the vertices added to $\text{REACH}(w_2)$ will be updated to contain the vertices in $\text{ADDS}(w_2)$. Suppose that there is some vertex, t, in $\text{ADDS}(w_2)$ but neither in $\text{REACH}(z)$ nor in $\text{ADDS}(w_1)$ — if this were the case, $\text{REACH}(z)$ would need to be updated again to add t. But by Lemma 6.6.4, $\text{ADDS}(z) \subset \text{ADDS}(w_1)$, and $t \notin \text{ADDS}(w_1)$ implies $t \notin \text{ADDS}(z)$, so no such t can exist. ■

Theorem 6.6.6 *The procedure AddEdgeTC-DAG in Figure 6.11 correctly updates the* REACH *sets for all vertices when an edge is added to a DAG, assuming that the added edge does not create a cycle.*

Proof From Lemma 6.6.3, we know that the only vertices that can need to be updated are the base of the added edge (x), and vertices whose immediate successors have changes in their REACH sets. For the base of the change, x, Lemma 6.6.2 assures that the set ADDS enqueued in step 12 is exactly the vertices that need to be added to REACH(x), i.e., DIFF(x). The first time through the main loop of phase 2, beginning at line 13, these vertices will be added to REACH(x) in lines 17–18. Since vertex x will never be encountered again (the only vertices added to the queue will be ancestors of x), REACH(x) is updated correctly.

When x is examined on the first pass through the main loop of phase two, the loop beginning at line 19 will examine each of the immediate predecessors of x; any immediate predecessor with a change in its REACH set will be enqueued in line 27. Inductively, since each time a vertex is dequeued its immediate predecessors will be considered, we see that if a vertex has an immediate successor whose REACH set changes, then it will be enqueued and tested. Hence the vertices identified by Lemma 6.6.3 will be considered once by the procedure. The VISITED set assures that no such vertex will be considered more than once; by Lemma 6.6.5 we know that the single visit is sufficient to correctly update the REACH set. ■

Detection of cycles

The AddEdgeTC-DAG procedure as given has a precondition that the addition of edge xy does not induce a cycle. We have enough information to detect edge additions that induce cycles: if x is in REACH(y), then there exists a path $y \rightsquigarrow x$ in the graph, so adding xy would create a cycle. We can test this precondition by adding the following line:

if $x \in$ REACH(y) **then return ERROR**

Unfortunately, for SSR-DAG we do not have adequate information to test the incoming edges, so if cycle detection is required, the AddEdgeTC-DAG procedure must be used.

6.6.3 Analysis of AddEdgeTC-DAG

Data structures for sets

The key operations in procedure AddEdgeTC-DAG in Figure 6.11 involve the sets ADDS, VISITED, and REACH. In order to achieve the desired complexity, we must be able to show that the following operations can all be implemented in time $O(1)$:

1. **initialize**(s): make s an empty set. (Line 3, implicitly, and Line 11.)
2. **add**(s, x): add item x to set s. (Lines 3, 18, 21, and 25.)
3. **in**(s, x): predicate to test whether $x \in s$. (Lines 2, 8, 20, 24)
4. **foreach**(s): generate the elements of set s (O(1) per set element generated). (Lines 7, 17, 19, and 23.)

To meet these requirements, a relatively clever data structure is required. The set must be implemented to support the **add** and **in** procedures in $O(1)$ time, without requiring $O(n)$ time to initialize, as would a standard bit vector implementation. Furthermore, we need to be able to list the members of a set in time proportional to the number of members, an operation not supported by a bit vector. A data structure with these characteristics can be found in, e.g., [LD91, pp. 258–259]. The amount of storage required to implement sets in this way will be larger by a constant factor that is approximately $2d$, where d is the number of bits per word for a particular machine. This is still $O(n)$, so from an asymptotic standpoint this makes no difference, but as a practical matter it may be smarter to implement the sets as bit vectors. Bit vectors require a factor of $O(n)$ for some of the set operations, but with very low constants and saving a substantial amount of space. For our complexity arguments we assume that the $O(1)$ implementation is used.

Definition of $\|\delta\|$

In order to analyze the algorithm in terms of $\|\delta\|$, we have to determine exactly what δ is in this case. Recall that δ = MODIFIED $\cup$ AFFECTED, where MODIFIED represents the set

of vertices that change directly as the result of the incremental update, and AFFECTED is the set of vertices with labels that change as a result. Since we are looking at single edge changes, MODIFIED is just the set $\{x, y\}$ for the addition or removal of an edge xy. Determining the AFFECTED set for transitive closure is more of a problem, since we are not updating a single label for each vertex, but rather a set of labels. A similar problem arose when Ramalingam and Reps analyzed All-Shortest Paths [RR91, p.15], and we use a similar, but not identical, approach. In order for our definition of AFFECTED to be reasonable, it must meet the spirit of δ-analysis, in which local changes require a small amount of work regardless of the input size of the start-over problem.

The DIFF sets defined above represent the changes made to the REACH sets at each vertex. Each element of DIFF(v) corresponds to one change that has to be made to the set REACH(v). Certainly, the amount of work done should be related to $\sum_{v \in V} |\text{DIFF}(v)|$, since this sum is the total number of changes made to all the REACH sets. Thus in this case rather than $\delta \leq n$, we have $\delta = \sum v \in V |\text{DIFF}(v)| \leq \sum v \in V n = n^2$. The problem is accounting for the neighborhood of the change — exactly which vertices should be included in counting the extended size of δ? The first answer would be that certainly every vertex v for which DIFF$(v) \neq \emptyset$ should be included. However, it would appear that this neighborhood alone is not enough. Consider updating a graph like the one shown in Figure 6.13. When the edge xy is added, the only REACH set update required is to

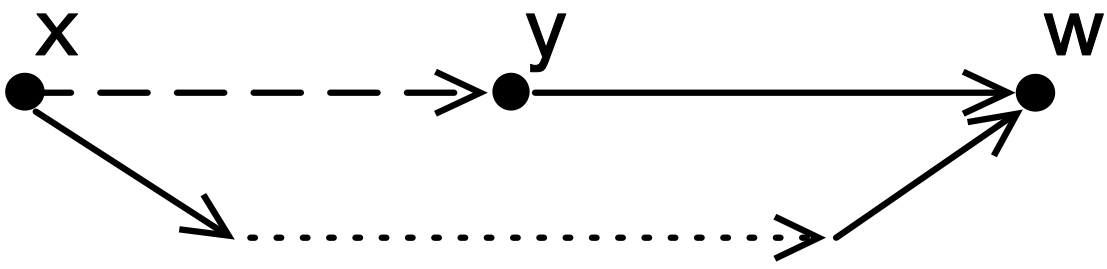

Figure 6.13: Sample update — dashed edge represents change, dotted edge represents an arbitrarily long chain of edges.

add y to REACH(x). How can it be determined that no other changes need be made to REACH(x)? Any straightforward algorithm, i.e., one that does not depend on any data structures other than the direct representation of the graph and the REACH sets, will have to use one of the following approaches:

1. Traverse all paths from x to descendants, without using edge xy. This is certainly not δ-incremental, since there is no bound on the length of the paths.

2. Compare the entries in REACH(y) with those in REACH(x). This would show, for example, that w is already in REACH(x) and therefore does not need to be added to REACH(x). The problem is that REACH(x) and REACH(y) both represent sets of size $O(n)$; "comparing them" amounts to taking the union of two, non-disjoint, sets, requiring $\Theta(n)$ time, and thus is not δ-incremental.

3. Take the approach in Procedure AddEdgeTC-DAG, by examining neighbors of y, *even though* REACH(y) — *the label of vertex* y — *is unaffected by the change.*

Our extension of the definition of δ is influenced by the third approach. For each change in a REACH set, say adding v to REACH(u), we allow the algorithm *to examine the neighborhood of the subgraph* $\{u, v\}$. Certainly it is reasonable to examine the neighborhood of u, and examining both vertices only doubles the number of vertices considered. The question is whether it is "fair" to count the neighborhood of a vertex v. We point out the following:

- If the density of the graph is uniform, then it makes no difference which vertex neighborhoods are searched.

- If the degree of the vertices is bounded, then the time to search the neighbors is bounded for all vertices, so it makes no difference which vertex neighborhoods are searched.

The only distortion would occur if the vertices whose REACH sets change have low degree, while the vertices in the DIFF sets have high degree. While it is certainly possible to construct examples like this, it is hard to imagine an environment in which this is generally the case, especially on average. If there is a situation in which this algorithm consistently searches high degree vertices when the changes are all in low degree vertices, the argument above suggests that we cannot expect to do much better anyway.

Definition 6.6.7 For the incremental update of a graph in which the label on a vertex is a set (or a vector) rather than a scalar, the change, δ, is:

$$\delta = \bigcup_{\{v \in V | \text{DIFF}(v) \neq \emptyset\}} v \times \text{DIFF}(v)$$

The size of the change, $|\delta|$, is simply the number of pairs in δ. The extended size of δ, $|\delta|$ is the sum of the extended sizes of the elements of δ; that is, if $p[1]$ represents the first element of a pair and $p[2]$ the second,

$$\|\delta\| = \sum_{p \in \delta} \|\{p[1]\} \cup \{p[2]\}\|.$$

The second order extended size[7] of δ is the sum of the second order extended sizes of the pairs in δ, i.e.,

$$\|\delta\|_2 = \sum_{p \in \delta} \|\{p[1]\} \cup \{p[2]\}\|_2.$$

If $x = p[1]$ or $x = p[2]$ for some $p \in \delta$, we say that x is an *affected vertex*.

We illustrate the definition with an example, using Figure 6.14. When edge xy is

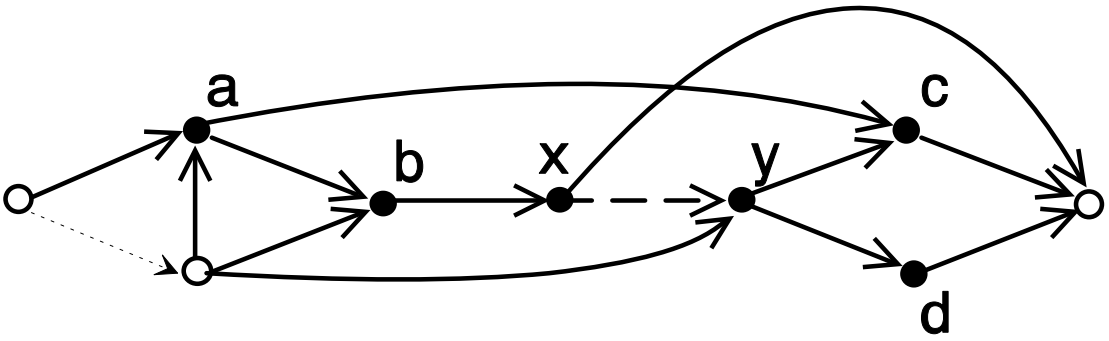

Figure 6.14: Illustration of Definition 6.6.7

added, the DIFF sets and δ are as follows:

$$\begin{aligned}
\text{DIFF}(x) &= \{y, c, d\} \\
\text{DIFF}(b) &= \{y, c, d\} \\
\text{DIFF}(a) &= \{y, d\} \\
\delta &= \{(x,y),(x,c),(x,d),(b,y),(b,c),(b,d),(a,y),(a,d)\} \\
|\delta| &= 8
\end{aligned}$$

[7] See §5.2 for the definition of the second order extended size.

In Figure 6.14, the affected vertices are black and the others are white. The neighborhood will include all the edges shown except the dotted one between two unaffected vertices. The extended size of δ will be determined by summing up the size of the neighborhoods for all the affected vertices:

$$\begin{aligned} \|\delta\| &= \|\{x,y\}\| + \|\{x,c\}\| + \|\{x,d\}\| + \|\{b,y\}\| + \\ &\qquad \|\{b,c\}\| + \|\{b,d\}\| + \|\{a,y\}\| + \|\{a,d\}\| \\ &= 6+6+5+7+6+5+8+6 \\ &= 49 \end{aligned}$$

Analysis of the algorithm

Theorem 6.6.8 *Procedure AddEdgeTC-DAG requires time* $O(\|\delta\|)$.

Proof In phase one, a vertex v is only enqueued, in lines 4 and 10, if it is not already in REACH(x), i.e., $v \in$ DIFF(x); furthermore, because of the test $v \notin$ ADDS in line 8, no vertex is added to the queue more than once. Hence the loop starting at line 5 is iterated once for each member of DIFF(x). Each iteration examines the successors of one vertex v, where $\{x,v\} \in \delta$, and hence the work in the loop at line 7 does no more work than $\|\{x,v\}\|$. Since all other operations require $O(1)$ time, the total time for phase one is bounded by

$$\sum_{v \in \mathrm{DIFF}(x)} \|\{x,v\}\| = O(\|\delta\|)$$

Phase two is similar except that now the predecessors of nodes with non-empty DIFF sets are examined. The VISITED set initialized in line 11, updated in line 21, and tested in line 20 assures that each such vertex is examined exactly once. Each time such a vertex is examined, the loop in lines 19–27 examines each predecessor, and performs one test, in line 24, for each element of DIFF(x). Thus each such test corresponds to a single pair $\{v,n\}$, where v is the vertex in line 19 and n the vertex tested in line 24. Thus phase two is bounded by

$$\sum_{\{u \mid \mathrm{DIFF}(u) \neq \emptyset\}} \left(\sum_{v \in \mathrm{DIFF}(u)} \|\{u,v\}\| \right) = O(\|\delta\|)$$

The total time required is the sum of the times for the two phases and hence is $O(\|\delta\|)$.

■

6.6.4 Edge Deletions for TC-DAG

Edge deletion for TC-DAG poses problems similar to deletion for SSR-DAG, but it is somewhat more complicated because we are now updating sets rather than simple flags. Addition is simple — if the edge addition has resulted in the addition of vertex z to the reach set for some successor v of u, then we can simply compute $\text{REACH}(u) \leftarrow \text{REACH}(u) \cup \{z\}$. If $z \in \text{REACH}(u)$ *before* the edge addition, then the union has no effect, but also causes no problem. But suppose instead that a v can no longer reach z as the result of a deletion. It would be a mistake to simply set $\text{REACH}(u) \leftarrow \text{REACH}(u) - \{z\}$, since there may be another path from u to z that does not pass through v. Thus before deleting z from $\text{REACH}(u)$ it must be determined that no such alternate path exists. Furthermore, this determination can only be made *after* all active successors of u are updated, just as in the SSR-DAG case (see Figure 6.8). The simple approach used for SSR-DAG is costly here, since each time a vertex is visited, a set union must be performed.

There are at least three approaches that might be taken to deal with the problem:

1. Maintain dynamically a priority order on the vertices in time $O(\|\delta^2\| \log \|\delta\|)$ [AHR+90], and test the vertices in reverse priority order. This guarantees that when a vertex v is checked, the REACH sets for all its successors are up-to-date. Note that in a strict sense this approach would not be δ-bounded for TC-DAG, since an incremental change might require no updating of the REACH sets and yet result in an unbounded amount of work to update the priority order. If, however, the maintenance of the priority order was needed for some other purpose, then it could be used.

2. Go ahead and update $\text{REACH}(u)$ each time u is a successor of an active vertex. Note that the error, if any, will always be conservative — a vertex will fail to be

deleted from REACH(u) that u can no longer reach, since it incorrectly remains in the REACH set for one of its successors. However, when the *last active successor of* u is processed, REACH(u) will be updated correctly. This yields the correct answer but could be costly since many vertices will be updated many times, each requiring a set union. This is the approach taken by SSR-DAG, but it is unnecessarily costly for TC-DAG.

3. Before processing a vertex u, check whether all its potentially active successors have been updated. A vertex $v \in$ SUCC(u) is a candidate for update only if $x \in$ REACH(v), where x is the base of the incremental deletion. A set, ACTIVE, will be maintained, listing the vertices that have been updated. Then vertex u would only be updated if $\forall v$ such that $v \in$ SUCC(u) and $x \in$ REACH(v), $v \in$ ACTIVE. This results in an implicit priority ordering of the updates. Note that it is possible that vertex u will never be updated, since one of its potentially active successors never becomes active; Lemma 6.6.4 justifies this.

We will use the last approach in procedure DeleteEdgeTC-DAG in Figure 6.16.

We begin by modifying subroutine CanReach. The modified CanReach(u, v, w) returns **true** if u can still reach an arbitrary vertex w *after* edge uv is removed, else returns **false**.

```
    CanReach(u, v, w)
1.      for each z ∈ SUCC(u) − {v} do
2.          if w ∈ REACH(z) then
3.              return true
4.      return false
```

Figure 6.15: Procedure to determine whether u could reach w after removal of edge uv.

Lemma 6.6.9 *Procedure CanReach*(u, v, w) *in Figure 6.15 correctly indicates whether vertex* u *could reach vertex* w *without using edge* uv*, assuming that the* REACH *sets for all successors of* u *are up-to-date.*

Proof If u can still reach w without using edge uv, it must be via a path $u, z, \ldots, w$, where $z \in \text{SUCC}(u) - \{v\}$. Note that edge uv cannot be in the path $z \ldots w$, because that would imply a cycle in the graph. Since CanReach checks each candidate successor and returns **true** if and only if there is a successor that can reach w, it works correctly. ■

Lemma 6.6.10 *Procedure CanReach(u, v, w) in Figure 6.15 requires time at most $O(\text{SUCC}(u))$.*

Proof CanReach has a single loop that is iterated once for each vertex in $\text{SUCC}(u) - \{v\}$; the operation in line 2 is $O(1)$. ■

Recall that $\text{DIFF}(v)$ refers to the change in $\text{REACH}(v)$ required after an incremental change. The following lemma is an analog to Lemma 6.6.4.

Lemma 6.6.11 *After an edge xy is deleted, $\text{DIFF}(v)$ will be a (not necessarily proper) subset of the* DIFF *sets for its successors that are ancestors of x.*

Proof Suppose there is some vertex t such that $t \in \text{DIFF}(v)$ and $t \notin \text{DIFF}(w)$, where $w \in \text{SUCC}(v)$ and $w \rightsquigarrow x$. Note that the only way vertex t can ever be in set $\text{DIFF}(v)$ is if $t \in \text{DIFF}(x)$, where x is the base of the edge added by AddEdgeTC-DAG, and there is a path $v \rightsquigarrow x$. Since $w \rightsquigarrow x$, by assumption, the deletion of edge xy removed a path $w \ldots xy \ldots t$; thus w cannot reach t by this path, so $t \notin \text{DIFF}(w)$ implies either that $t \notin \text{REACH}(w)$ *before* the incremental change, or that there is an alternate path $w \ldots t$ that does not use xy. In the first case, $t \notin \text{REACH}(w) \Rightarrow t \notin \text{REACH}(v) \Rightarrow t \notin \text{DIFF}(v)$. In the second case, there is a path $vw \ldots t$ that does not use xy, and hence $t \notin \text{DIFF}(v)$.

■

Corollary 6.6.12 *If vertex v has a successor z that is an ancestor of x, and z is not active, then v is not active.*

```
    DeleteEdgeTC-DAG(x, y)
        /* phase one */
1.      emptyQueue(q)
2.      if not CanReach(x, y, y) then
3.          DELS ← {y}
4.          enqueue(q, y)
5.          while not isEmpty?(q) do
6.              dequeue(q, z)
7.              for each v ∈ SUCC(z) do
8.                  if v ∉ DELS & not CanReach(x, y, v) then
9.                      DELS ← DELS ∪ {v}
10.                     enqueue(q, v)
11.         enqueue(q, (x, DELS))
12.         ACTIVE ← {x}
        /* phase two */
13.     while not isEmpty?(q) do
14.         dequeue(q, p)
15.         z ← p[1]
16.         SUCCDELS ← p[2]
17.         for each n ∈ SUCCDELS do
18.             REACH(z) ← REACH(z) − n
19.         ACTIVE ← ACTIVE ∪ {z}
20.         for each v ∈ PRED(z) do
21.             READY ← true
22.             for each t ∈ SUCC(v) − {z} do
23.                 if x ∈ REACH(t) & t ∉ ACTIVE then
24.                     READY ← false
25.             if READY then
26.                 DELS ← ∅
27.                 for each n ∈ SUCCDELS do
28.                     if not CanReach(v, z, n) then
29.                         DELS ← DELS ∪ {n}
30.                 if DELS ≠ ∅ then
31.                     enqueue(q, (v, DELS))
```

Figure 6.16: Procedure for incremental edge deletion from a DAG

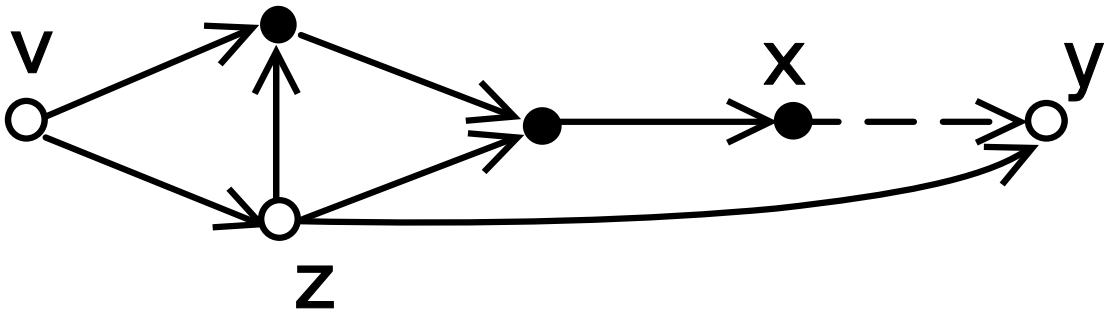

Figure 6.17: Example illustrating Corollary 6.6.12.

Proof An example illustrating this corollary is in Figure 6.17. Since by Lemma 6.6.11, DIFF(v) is a subset of the DIFF sets for its successors that are ancestors of x, and DIFF(z) = {}, DIFF(v) = {}; thus v is not active. ■

Lemma 6.6.13 *When the set* DELS *is created in lines 26–29, the successors of v are up-to-date.*

Proof By Lemma 6.5.2, any vertex $z \in$ SUCC(v) such that $z \not\rightsquigarrow x$ will not be active, and hence is up-to-date. So we need be concerned with only those successors of v that might become active, but have not yet been updated. The value of READY will be set to **false** in line 24 if there is any such successor; hence line 26 is reached only if every potentially active successor has been updated, and thus every successor of v is up-to-date. ■

Theorem 6.6.14 *Procedure DeleteEdgeTC-DAG in Figure 6.16 correctly updates the* REACH *sets in a DAG after the deletion of an edge xy.*

Proof The loop in lines 17–18 is the only place that the REACH sets get updated; REACH(v) will be updated if and only if v is enqueued, hence if v is active. If a vertex is not the base of the change, x, and does not have an active successor, it will never become active, and hence its REACH set remains as before, as justified by Lemma 6.6.3.

The REACH sets are updated using the set SUCCDELS, which when enqueued in line 31 was set DELS. Lemmas 6.6.9, 6.6.13, and 6.6.11 assure us that SUCCDELS contains

exactly the vertices in DIFF(z), so if z is updated it is done correctly. If a vertex v has one or more active successors, then it will be enqueued in q only if it has an active successor that is processed *after* all other potentially active successors of v have been processed, by Lemma 6.6.13. If v has a potentially active successor that does not become active, then it may not be processed, but by Lemma 6.6.12 this is not a problem since v must be inactive. ■

6.6.5 Analysis of DeleteEdgeTC-DAG

We use the definition of δ, $\|\delta\|$, and $\|\delta\|_2$ in Definition 6.6.7.

Theorem 6.6.15 *Procedure DeleteEdgeTC-DAG in Figure 6.16 requires* $O(\|\delta\|^2) + O(\|\delta\|_2)$ *time per deletion.*

Proof Note that all set operations are $O(1)$ as discussed above.

If the deletion of xy does not remove any vertex from REACH(x), then the only work performed by the algorithm will be the call to CanReach in line 2, which looks at the neighborhood of x. Otherwise, Phase one adds a vertex to q only if it is in DIFF(x). For each vertex v that is added to q, there is a call to CanReach that looks at each successor of v; thus the number of calls to CanReach is bounded by $\|\text{DIFF}(x)\| = O(\|\delta\|)$. Each call to CanReach looks at every successor of x; $O(|\text{SUCC}(x)|) = O(\|\delta\|)$. Hence phase one is $O(\|\delta\| \times \|\delta\|) = O(\|\delta\|^2)$.

In phase two, a vertex z dequeued in line 14 could only have been added to q, in line 11 or line 30, if DELS $\neq \emptyset$, i.e., DIFF(z) $\neq \emptyset$. Each $O(1)$ operation in line 18 will correspond to one of the members of δ. The loop in lines 20–31 is executed for predecessors of an affected vertex. The first inner loop, in lines 22–24, examines successors of these predecessors; hence the number of vertices is examined in line 23 is bounded by the second order extended size of $\delta = O(\|\delta\|_2)$. Finally, the second inner loop calls CanReach once for each element of δ, and each call requires time $O(\text{SUCC}(v)) = O(\|\delta\|)$. Thus the total for phase 2 is $O(\|\delta\|_2) + O(\|\delta\|^2)$. ■

6.6.6 Comments

In the previous pages we have presented an incremental algorithm for updating the transitive closure of a DAG, where additions require time bounded by $O(\|\delta\|)$ and deletions $O(\|\delta\|^2) + O(\|\delta\|_2)$. Typically, algorithm performance is analyzed as a function of the input size rather than as a function of δ. In this section, we consider the performance of these algorithms under different assumptions about the DAG.

Let $n = |V|$ and $m = |E|$. Observe that δ can be as large as $O(n^2)$ (see §6.6.3) and hence $\|\delta\| = O(n^3)$; it is easy to envision incremental additions or deletions that change, say, half the elements in the REACH set for every vertex in the graph. Hence the overall complexity of AddEdgeTC-DAG is $O(n^3)$ in the worst case.

At first glance, DeleteEdgeTC-DAG would appear to be as bad as $O(n^6)$, since $\|\delta\|^2$ could be as big as n^6. However, examining the worst case performance of DeleteEdgeTC-DAG directly reveals that things are not quite that bad.

Theorem 6.6.16 *The worst case complexity of Procedure DeleteEdgeTC-DAG in Figure 6.16 is* $O(n^3)$.

Proof Each call to CanReach is $O(n)$ since it examines the successors of some vertex. Phase one can examine at most $O(n)$ vertices; for each vertex v examined, CanReach is run once for each element of SUCC(v). Thus the total work performed by phase one is $O(n^3)$.

Similarly, in phase two, at most $O(n)$ vertices can be dequeued. The loop in lines 17–18 is $O(n)$. The loop in lines 20–31 will be executed $O(n)$ times; each execution requires a $O(n)$ loop at lines 22–24. The loop at lines 27–29 requires $O(n)$ iterations of the $O(n)$ CanReach procedure. Note however that this loop can be executed at most *once for each vertex* — a vertex cannot be "ready" more than once. Thus *all* the executions of lines 27–29 require $O(n^3)$.

Since the worst case complexity of both phases is $O(n^3)$, the desired complexity results. ■

When processing a graph of bounded degree, the incremental procedures are guaranteed to be somewhat faster.

Theorem 6.6.17 *For a graph with degree bound d, the worst case complexity of Procedure AddEdgeTC-DAG in Figure 6.11 is* $O(dn^2)$.

Proof Phase one requires the examination of $O(n)$ vertices, and for each a visit to $O(d)$ successors, or $O(dn)$ total.

Phase two can visit $O(n)$ vertices, with $O(n)$ steps at lines 17–18. The loop in lines 19–27 will be executed $O(d)$ times; the inner loop at lines 23–25 is $O(n)$. Thus phase two is $O(dn^2)$, which is also the complexity of the entire procedure. ■

Theorem 6.6.18 *For a graph with degree bound d, the worst case complexity of Procedure DeleteEdgeTC-DAG in Figure 6.16 is* $O(d^2n^2)$.

Proof Procedure CanReach depends on the successors of a vertex, hence it requires $O(d)$. Phase one examines $O(n)$ vertices; for each one, it examines $O(d)$ successors, running the $O(d)$ CanReach procedure on each. Hence phase one is $O(d^2n)$.

In phase 2, again we find that $O(n)$ vertices are examined. The dominant complexity is the loop at line 27 which can be executed $O(n)$ times; each execution requires $O(n)$ iterations of CanReach. Thus phase two requires $O(dn^2)$.

The total complexity for the procedure is $O(d^2n) + O(dn^2) = O(d^2n^2)$. ■

The advantages of an algorithm bounded by δ is that it is guaranteed to work efficiently when δ is small. Consider changes of size $O(1)$. In this case, the cost for an AddEdgeTC-DAG, δ, will really depend upon the size of the neighborhood of the change. In some situations in which an incremental algorithm for a DAG is applied, the graph may have bounded degree [Yel91]. In this case, the size of the neighborhood is also bounded by a constant, and the cost of AddEdgeTC-DAG is also $O(1)$. In general, the cost could be $O(n)$, since if there is no bound on the degree of the graph, the

neighborhood could potentially be as big as n. Since DeleteEdgeTC-DAG is bounded by a higher function of δ, the same argument holds, just with a higher constant.

While the asymptotic worst case cost of the δ-incremental algorithm is the same as the asymptotic worst case cost of recomputing from scratch, it is possible that the cost of running DeleteEdgeTC-DAG could be worse than recomputing TC-DAG. However, it is never necessary to spend *more* than twice the cost of recomputing from scratch — the two algorithms (start over and incremental) can be dovetailed, with work terminating when the first algorithm is completed. Furthermore, like all δ-bounded incremental algorithms, the benefits will be greatest when δ is small. If many updates are localized, that is, have small δ's, and if the average degree of the graph is small, this algorithm is likely to have substantially better performance than starting over. While a general description of the typical problem graph can suggest whether or not this algorithm will be efficient, empirical testing is probably the only way to determine which problem sets will lend themselves to this algorithm.

Chapter 7

Analyzing Incremental Algorithms — What Do We *Really* Want to Know

7.1 What is an Incremental Algorithm?

The terms "incremental algorithm," "dynamic algorithm," "update algorithm," and "on-line algorithm" all refer to algorithms in which a solution is maintained or modified as a result of a stream of data. Different authors have used different definitions for the above terms, and it seems clear that there is no consensus on what an incremental algorithm is or what it should do. Our Definition 2.1.1 in Chapter 2 is intentionally restrictive: we consider only "atomic" changes, not sequences.[1] In this section we consider two basic flavors of incremental algorithm; in subsequent sections, we discuss the types of analysis likely to be most appropriate for each. As we shall see, the distinctions are as much a matter of emphasis as of definition, and there are some algorithms that defy easy categorization. However, we believe that the categorization we present makes a useful start at understanding the ways in which incremental algorithms are used.

7.1.1 Update Algorithms

Most of the algorithms that we consider in this dissertation can be conveniently classified as update algorithms. That is, there is some problem that can be solved by some standard algorithm to find a solution; the goal is to find an algorithm that, given a small change in the input, computes the new answer faster than starting over from scratch. Definition 2.1.1 approaches the problem from this point of view. Typically,

[1] Despite this, the results do have implications for sequences; see §3.3 and §7.2.3.

the Holy Grail of this approach has been to try to find an incremental algorithm with better worst-case performance than the start-over algorithm. Examples of this approach include Frederickson's algorithm for updating minimum spanning trees [Fre85], which has guaranteed improved worst-case performance over the start-over algorithm, and Ryder's algorithms for incremental data flow analysis [RP88], which in the worst case is no better than starting over. Update algorithms that beat the start-over algorithm for a single change, in the worst case, have been relatively few; the IRLB method gives some insight into why this has been the case for many problems.

Perhaps as a result of the difficulty of finding incremental algorithms that beat starting over in the worst-case, alternative methods of characterizing the performance of incremental algorithms have been proposed. First, even if the incremental algorithm can be as bad as starting over, it may be that the "expensive cases" to update are relatively rare, and hence the algorithm has good average case performance. Or, even if the average case is of the same order of complexity as starting over, it may be that the constants are smaller. It is clearly difficult to find a convincing abstract model for the mix of inputs that will be updated incrementally, and we know of no case in which a complex incremental algorithm has been analyzed for average case complexity. For some domains, it may be possible to compare the utility of incremental algorithms considering their performance for the types of inputs that are most likely to occur [BR90]. There have also been a few attempts to demonstrate the utility of incremental algorithms by comparing their performance on random data, versus starting over from scratch; see [Car88, RLP90]. While it has often been suggested that a better empirical test would be based on data collected from actual problem domains, we are not aware of any work of this kind.

Second, some problems may have the characteristic that for some incremental changes it may be possible to update the answer by looking at a relatively small part of the input set. If it is possible to characterize the work that "has to be done" after a change, this gives a lower bound on how well the incremental algorithm can perform. The goal then becomes to bound the work of the algorithm by this obligatory amount. Since the work that "has to be done" might, in some cases, be as much as was done

to solve the problem from scratch, the worst-case performance may be no better, but then we could not expect it to be. This is the motivation behind δ-analysis.

It is often suggested that amortized analysis [Tar84] may be a useful way to compare incremental algorithms. However, we would argue that in the update model such methods often are neither particularly appropriate nor useful. The appropriateness of amortization depends upon a concern with the overall running time of a sequence of operations; as Tarjan writes, "In many uses of data structures, a sequence of operations, rather than just a single operation, is performed, and we are interested in the total time of the sequence, rather than in the times of the individual operations." That is not necessarily the case here. If our incremental algorithm is going to update data flow information, as part of a programming environment, clearly the user cares how quickly the environment can update itself after a single change. The fact that some sequence of changes will be handled quickly is a lesser concern. The user may well prefer an algorithm that always runs adequately fast to one that runs faster in an amortized sense but takes a long time every so often.[2] For any algorithm that can be viewed as real-time, it will be the worst-case performance that will be most crucial. This is not to say that an amortized argument, if available, is not interesting; but rather it is not as interesting as in some other cases, as discussed in the next section. As a final comment, we have observed that amortization arguments often depend on a sequence of changes of the same kind, e.g., adding elements, in which a data structure "moves in one direction." An example is the disjoint set union problem [Tar75], in which the "entropy" of the data structure can be shown to decrease as set lookups are performed. Fully dynamic problems (see §7.2.1) are harder to bound with an amortized argument, because there may be an expensive case that can be reached many times as a result of a sequence of additions and deletions.

[2]The frustration of waiting for garbage collection comes to mind here.

7.1.2 Online Algorithms

A standard, start-over algorithm can be viewed as an *offline algorithm*: the algorithm starts with the entire input set in memory, and can manipulate this input in any way to improve the efficiency of the algorithm. For example, consider Heapsort [AHU74]: before the algorithm can sort, it must put the input items into a heap. Not until long after the entire data set has been input can we expect any output. Alternatively, an *online algorithm* executes by reading its input an item at a time and computing as it goes. Aho, Hopcroft and Ullman define online execution, for an input sequence ρ, as follows:

> **Definition.** The *on-line* execution of ρ requires that the instructions in ρ be executed from left to right, executing the ith instruction in ρ without looking at any following instructions. The *off-line* execution of ρ permits all of ρ to be scanned before any answers need to be produced. [AHU74, p109]

Note that this definition is given in terms of a sequence of "instructions." While there is no *fundamental* difference between instructions and any other kind of input data, the description of the input as a sequence of instructions is typical for an online problem.

For online problems, the most effective approach has often been to develop algorithms with a good amortized complexity. As Tarjan writes:

> A worst-case analysis, in which we sum the worst-case times of the individual operations, may be unduly pessimistic, because it ignores correlated effects of the operations on the data structure. On the other hand, an average-case analysis may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false. In such a situation, an amortized analysis, in which we average the running time per operation over a (worst-case) sequence of operations, can yield an answer that is both realistic and robust. [Tar84]

There is no hard and fast line between algorithms we call "update" and those we call "online." It is more a matter of the questions that we choose to ask. Online algorithms typically refer to a sequence of operations, rather than a solution that needs to be updated. Online algorithms are often associated with a particular data structure. Finally, with online algorithms we are generally interested in a sequence of operations, rather than a single update. With these notions in mind, we will examine the approaches to incremental algorithms considered in this dissertation.

7.2 Considerations in the Analysis of Incremental Algorithms

Finding lower bounds for incremental problems is inspired by the general problem of finding lower bounds. Of course, an incremental or dynamic problem can be modeled in the same way as any other problem, and the same techniques applied. However, the particular nature of incremental problems has made this a difficult and often fruitless task. Thus we are led to explore the techniques discussed in this dissertation. In this section we will address some of the special issues that pertain to incremental algorithms.

7.2.1 Fully Versus Partially Dynamic

In the typical incremental problem, the incremental changes applied are categorized as *additions* or *deletions*. In graph problems, this corresponds to adding an edge or deleting an edge.[3] Changes in weights in a weighted graph generally correspond to additions and deletions. If the problem goal is to minimize some set of weights, as in the minimal spanning tree and shortest path problems, an increased weight can trigger the response associated with a deletion, and a decreased weight corresponds to an insertion.

Incremental algorithms can be categorized as *fully-* or *partially*-dynamic, depending on whether they handle a single type of change, e.g., insertions, or both insertions and deletions.[4] For example, Tarjan's "Union-Find" algorithm [Tar75] can be viewed as a partially-dynamic incremental algorithm for the Connected Components problem; the algorithm handles edge additions, but apparently cannot be generalized to handle deletions. Frederickson's algorithm for updating minimum spanning trees [Fre85], on the other hand, is an example of a fully-dynamic algorithm. Edge additions and deletions, as well as weight increases and decreases are all addressed. While for some domains it is useful to develop partially-dynamic algorithms, clearly the more general, and often

[3]Note that if it is necessary to model adding or deleting vertices in the graph, such vertices should have no incident edges when the change is made. In our view, the addition or deletion of a vertex with multiple edges should be viewed as the incremental addition or deletion of each individual edge. This is especially true when considering worst-case performance; for amortized arguments, it may make sense to treat a sequence of changes as a single unit.

[4]In some cases, the algorithm may handle both types of change, but the analysis may apply only to a sequence of one type of change; see, e.g., [Yel91].

more challenging, goal is finding a fully-dynamic algorithm. Typically, although not always, online algorithms are partially-dynamic.[5]

Likewise, a lower-bound argument can apply to partially or fully dynamic versions of the underlying problem. Note that if a lower bound can be found *either* for additions or deletions, the bound applies to the fully-dynamic version of the problem. For example, most of the IRLB arguments given in Chapter 4, and two of the three δ-analysis arguments in Chapter 5, apply specifically to the deletions case; as such, they represent lower bounds for the fully-dynamic problem, since a fully-dynamic algorithm must address deletions. However, such bounds do not apply to partially-dynamic algorithms that perform additions only.

In most cases with which we are familiar, deletions are the harder problem. Take, for example, the connected components problem, shown in §4.4.2 to be in IRLB Class 2. Since connected components can be solved non-incrementally in $O(n+m)$, i.e., linear time, the IRLB result implies that for a dense graph, a fully-dynamic algorithm cannot be faster than $O(\sqrt{m})$. Furthermore, a partially-dynamic algorithm that handles deletions only is also bounded by $\sqrt{m}$. On the other hand, the Union-Find algorithm can be used to update a sequence of additions in amortized time $o(\sqrt{m})$ per addition. In this case, it is clear that updating after deletions is inherently more difficult than updating after insertions.

However, deletion is not always the more difficult problem. Consider the problem of maintaining a topological sort labeling τ (see §6.3.2). When a new edge xy is added, the condition $\tau(x) < \tau(y)$ will not necessarily hold; an incremental algorithm will have to relabel some of the vertices to bring the sort up to date. However, the deletion of any edge cannot violate the topological sort condition; for this problem, deletions are trivial.

We note that it may be possible to find partially-dynamic incremental algorithm for the "hard" direction that is more efficient, or simpler, than a fully-dynamic incremental algorithm. For example, the method of Even and Shiloach for updating connected

[5]List maintenance problems are a notable exception; see, e.g., [ST85].

components requires $O(n)$ per deletion (amortized) [ES81]. By working in only one direction, a partially-dynamic algorithm can "throw away" structure as it goes, whereas a fully-dynamic algorithm typically needs to maintain complicated history information.

Finally, we note that there has been quite a bit of work on persistent algorithms, which can be thought of as partially-dynamic algorithms that can back up to previous states. See, for example, [Sar86, Wes89].

7.2.2 Model of Computation

A lower bound is most applicable when it is proved for a robust model of computation. The most appealing model for sequential lower bound proofs is the *Random Access Machine (RAM)* [AHU74], which closely models a typical computer. Not all incremental lower bounds are derived in the RAM model. The well-known super-linear lower bound for the Union-Find algorithm is proved for a *Pointer Machine* [Tar75]; in this model all memory accesses follow via pointers, with no address arithmetic allowed. The lower bounds discussed in Chapter 5 are based on a model called *Local Persistence*; this is similar to a pointer machine in which the allowed pointers are required to match the structure of the problem being solved.

7.2.3 Worst Case Versus Amortized Results

Most lower bounds for incremental problems are bounds on the worst case performance of an algorithm. In some environments, an algorithm with good amortized performance may be acceptable, even if some steps may have a poor worst case. As discussed above, such results are particularly natural for online problems. For example, the Union-Find algorithm can require linear work for a single update step; yet the amortized cost per update is just slightly more than constant ($\alpha(m, n)$).[6] This suggests that it would be desirable to develop bounds that would apply to the amortized cost of incremental algorithms.

The general bound for incremental algorithms, Theorem 3.2.3 in Chapter 3, also

[6] $\alpha(m, n)$ is the functional inverse of Ackerman's function; m is the number of operations, and n is the number of sets unioned.

applies to amortized algorithms, as shown by Corollary 3.3.1. The basic idea behind the proof to Theorem 3.2.3 is to show that running an incremental algorithm once for each item in the input set has the same effect as solving the problem from scratch; thus, we argue, the speed of the incremental algorithm cannot be faster than 1 over the problem size $(1/\ell)$, in the worst case. To see the application to amortized analysis, consider that we are talking about a sequence of ℓ calls to the algorithm, and hence we not only have a worst-case bound, but a bound on the average performance per operation.

The applicability of IRLB bounds to amortized analysis varies inversely with the tightness of the bound. Problems listed in Class 2 in Chapter 4 apply to graphs with n vertices and m edges; for these problems, the performance of an incremental algorithm is bounded by $1/n$. In this case, we are considering a sequence of n executions of the incremental algorithm, where it may be the case that $n \ll \ell$. Hence the IRLB bound is also an amortized bound for a sequence of length n. Unfortunately, the tightest IRLB bounds (Class 3) have nothing to say about amortized complexity because they are based on a single worst-case execution of the incremental algorithm. Also, there have not been any results giving incremental lower bounds in the δ-analysis model; it is not clear that this model lends itself to an amortized approach.

7.2.4 Relative Versus Absolute Bounds

The bounds presented in Chapters 3 and 4 are *relative*; they do not, by themselves, state an absolute lower bound for any particular algorithm. Rather, they bound the performance of the incremental algorithm as a function of the best possible non-incremental algorithm. The δ-analysis bounds in Chapter 5, on the other hand, are absolute bounds.

Where there is a known tight lower bound for the non-incremental algorithm, a hard lower bound for the incremental version can be immediately derived. Take, for example, the problem of sorting by comparisons. Since $\Omega(\ell \log \ell)$ is a proven lower bound for the start-over problem, $\Omega(\log \ell)$ is a hard lower bound for the incremental problem. As another example, consider the minimum spanning tree problem. Clearly this problem has a naïve linear lower bound $\Omega(m+n)$; the upper bound is $O(m + n \log n)$ [CLR91].

For a dense graph, $m = O(n^2)$, hence $O(m + n \log n) = O(n^2 + n \log n) = O(n^2)$, i.e., linear. Since minimum spanning tree is a Class 2 problem, it cannot be updated faster than $1/n$. Thus, for a dense graph, i.e., $m \approx n^2$, there is a firm lower bound of $\Omega(\sqrt{m})$ for updating minimum spanning tree. For these cases, the relative lower bound is as good, or nearly as good, as an absolute bound.

7.3 A Critique of IRLB's

In light of the considerations discussed in §7.2, we now examine the strengths and weaknesses of the IRLB approach to lower bounds.

7.3.1 IRLB's May Not Apply to Partially Dynamic Methods

An IRLB proof is based on a sequence of incremental operations. In all of the proofs presented in Chapter 4, this was a sequence of one or more deletions. Such a proof represents a bound for a fully-dynamic algorithm, since the fully-dynamic algorithm must handle any sequence of insertions or deletions. Furthermore, since the proof is based on deletions, it also applies to partially-dynamic algorithms that process deletions. However, an IRLB based on a sequence of deletions says nothing about sequences of additions only. This was pointed out in [Cul91]. It is not clear that this should be considered a limitation of the method; rather, it may reflect the fact that for many incremental problems, one side of the incremental problem is more difficult than the other, as discussed in §7.2.1.

There is a substantial amount of interest in fully-dynamic incremental algorithms. Frederickson's algorithm for handling updates to Minimum Spanning Trees and Connected Components stands out as the best-known result in this field, but others have made progress as well. For example, Italiano's dissertation [Ita91] presents fully-dynamic algorithms for updating 2-Edge Connectivity. Italiano also notes:

> . . . for many graph theoretical problems there is no known efficient partially dynamic algorithm for handling edge deletions only . . . . Since . . . dynamic problems with edge insertions only can be solved much more efficiently than the corresponding fully dynamic problems, it seems quite natural

> to ask whether more efficient techniques for handling edge deletions only can be designed. [Ita91, p. 95]

The interest shown in developing fully-dynamic incremental algorithms justifies lower bound methods, such as the IRLB, which are based on sequences of deletions, or additions, whichever is the more difficult case. In addition, there may be problems for which unmixed sequences of either deletions or additions may be efficient, at least in the amortized sense, but where it is still difficult to obtain an efficient, fully-dynamic incremental algorithm. This was the case for Transitive Closure on a DAG prior to the algorithm in Chapter 6.

7.3.2 Restrictions on Applicability — Initialization

It is possible to construct algorithms for particular problems that violate the initialization assumption, and hence for which no bounds can be found. However, when considering the incremental update algorithms that have been published, all meet the initialization requirements. This restriction seems to be more a theoretical issue than a practical one.

Clearly there is a trade-off. If an arbitrary amount of initialization and space is permitted, extremely fast incremental algorithms can be constructed. One can imagine constructing a huge table containing every possible answer to a problem, indexed by some signature of the input. Consider connected components in the form of a decision problem: given a graph, return 1 if the graph is connected, else return 0. Let

$$N = \begin{pmatrix} n \\ 2 \end{pmatrix}$$

The graph's signature can be formed by a string of N 0's and 1's, where 1's represent edges in the graph, represented in a natural order. Let the vertices of the graph be labeled $1 \ldots n$, and let the edge between two vertices be indicated by the labels of the two vertices it connects; e.g., the edge between vertex 1 and vertex 2 is 12. Then the bit signature for the graph could be organized as:

$$12\ 13\ \ldots\ 1n\ 23\ 24\ \ldots\ 2n\ 34\ \ldots\ (n-2)n\ (n-1)n$$

For example, a three-vertex graph might have the signature 011, indicating that the graph includes edges 13 and 23 but not edge 12.[7] For graphs of n vertices, construct an N-dimensional array A of 0's and 1's. Each dimension represents one bit in the signature, and hence is indexed from 0 to 1. The array is initialized by solving the connected components problem for the 2^N possible graphs with n vertices, and recording a 0 or 1 as appropriate; this algorithm would require $n2^N = O(n2^{n^2})$, which is prohibitively large. To begin the incremental process, the initial index into the array is computed, in time $O(n^2)$. After each incremental change, the output can be updated in constant time by computing the change in the offset induced by the change of a single index in the offset equation.

Such an algorithm bypasses the $\Omega(\sqrt{m})$ lower bound for fully dynamic connected components derived via IRLB analysis. Clearly, the requirements for Theorem 4.2.4, which require that the time spent by initialization be dominated by the time required to solve the problem from scratch, have been violated. However, an algorithm that requires exponential space and preprocessing time to quickly solve an incremental problem, when the non-incremental solution is linear, is of no practical interest. Our analysis does not rule out the possibility of a tighter trade-off; for example, could an initialization requiring quadratic preprocessing beat the IRLB for connected components? This is an open question. We believe that by focusing attention on initialization, the IRLB may suggest fruitful avenues for algorithm development.

7.3.3 Relative Lower Bounds

IRLB's are by their nature relative; that is, they bound the complexity of an incremental algorithm in terms of the time required for the start-over algorithm. We argue that this is not a significant deterrent to the value of the bound in most cases. First, many of the problems for which there has been interest in incremental methods, such as Connected Components and Transitive Closure (see, e.g., [YS88]), have linear time start-over algorithms; in this case, the lower bound is also known (linear) and hence an

[7] The reader may recognize that the signature corresponds to the upper-diagonal portion of an adjacency matrix.

IRLB can be used to find an absolute lower bound on the incremental algorithm.

At the other extreme, Theorem 3.4.1 suggests that, assuming $P \neq NP$, it is unlikely that there are any incremental algorithms for NP-complete problems with good worst-case bounds; in fact, finding a fast incremental algorithm for such a problem would be equivalent to proving $P = NP$, and finding an exponential absolute lower bound would show that $P \neq NP$. It is clear in this case we are not likely to do better than finding a relative bound.

Many of the problems in between have known lower bounds that are tight or close to tight. For example, sorting and minimum spanning tree are addressed in §7.2.4. For some other problems, e.g., shortest paths, the larger gap between the lower and upper bounds would make an absolute bound on the incremental algorithm more interesting.

7.4 A Critique of δ-Analysis

For certain types of problems, δ-analysis gives some insight into the inherent difficulty of incremental updates. However, this form of analysis, and particular lower bounds derived in the local persistence model, have some important limitations. In this section we discuss the uses and limitations of δ-analysis.

7.4.1 Algorithms with Bounded δ Complexity May Have Poor Worst-Case Performance

δ-analysis considers the performance of an algorithm as a function of the number of graph vertices that must be touched (δ). An algorithm is said to be *bounded* if the performance can be stated as a function of δ (§5.2). In one sense this is an appealing measure because it requires that the complexity of the algorithm must be related to the amount of work that, in some sense, *has* to be done. However, bounded does not necessarily mean efficient. Since nearly all incremental problems have updates that require changes of size $O(n)$, a complexity that is exponential in δ implies that the cost to process a single update could be exponential in n. For a problem with polynomial complexity, this type of incremental update is unlikely to be useful. Even for changes

with small δ, if the function is bad enough then the update algorithm will not be useful.

Furthermore, the performance algorithms do not really depend only on δ, but rather on the size of the neighborhood associated with δ, that is, $\|\delta\|$. This issue is discussed in §5.2.1. Thus there is an implicit dependence on the density of the graph — when the number of edges is bounded, or low on average, δ-incremental algorithms will be more attractive than when many vertices have $\Omega(n)$ neighbors.

One should point out that while these are limitations of the general approach, in practice many of the algorithms proposed have functions that are not only bounded, but also of quite reasonable complexity [Rep82, RR91, AHR+90]. Furthermore, large graphs from domains such as data flow analysis are likely to have low average density, even when there is no *a priori* bound on the degree of the graph. These points suggest that δ-incremental algorithms offer the promise of good performance for many practical problems.

7.4.2 Incremental Algorithms for Problems of Complexity $\omega(n)$

Consider the problem of graph connectivity, as discussed above in §5.4.1. As shown by Theorem 5.4.3, this problem is non-incremental in the local persistence model. This result means in effect that even though some changes in the graph may require updating only a constant number of vertices, any algorithm for the problem will visit an unbounded number of vertices, assuming the algorithm meets the restrictions of the local persistence model of computation. However, the complexity of computing this result from scratch depends not on the number of *vertices* in the graph, but on the sum of the number of edges (n) and the number of vertices (m), i.e., $O(n+m)$. When m grows faster than n, as it does in all but sparse graphs, $n+m$ will be $\omega(n)$, i.e., the complexity of the problem is strictly greater than n. An incremental update algorithm that had a worst-case performance of $O(n)$ could be quite good; in fact, an $O(n)$ lower bound on this problem can be derived from the IRLB argument for connectivity (see §4.4.2). Furthermore, $O(n)$ is the performance of Frederickson's algorithm on dense graphs [Fre85].

As the inherent complexity of the problem increases, a δ-analysis lower bound becomes progressively less interesting. For example, consider path-finding, or closed-semiring problems, which are conjectured to require $\omega(n^2)$ time, although the only known bound is $\Omega(n^2)$ [CLR91]. The all-pairs shortest paths problem is a special case of a closed-semiring problem. By IRLB analysis we know that incremental algorithms for this problem can be no faster, in the worst case, than starting over (see §4.4.1). Thus, we know that it is impossible to find an incremental algorithm that does better than $O(n^2)$. Via δ-analysis it can be shown that closed-semiring problems are non-incremental in the local persistence model [RR91]; however, this is weaker than the more general IRLB bound, and hence is not particularly useful. For any problem with complexity $\omega(n^2)$, the general lower bound (Theorem 3.2.3) suggests that the complexity of an incremental update will be $\omega(n)$; in such cases, showing the problem to be non-incremental in the δ-analysis problem does not yield a better lower bound.[8]

7.4.3 Weakness of the Local Persistence Model of Computation

At first glance, the local persistence model (§5.3.1) looks fairly close to the *pointer machine* model [Knu73]. The pointer machine, while strictly weaker than the *random-access machine* model of computation [AHU74], is a realistic model for list-processing algorithms [Tar83]. The fundamental difference between the pointer machine and the random access machine is that, in the pointer machine, address arithmetic is not allowed. The pointer machine model was used to derive the lower bound for the Union-Find algorithm [Tar75].

However, local persistence is substantially weaker than a pointer machine, because of the restriction that the pointers followed during a computation must follow the structure of the input problem. In other words, no data structures can be used to assist in the update. While this is not strictly the case, since there is no restriction on the data structures that might be used *within* a single vertex, the prohibition of pointers to other vertices except for neighbors in the underlying graph eliminates the methods

[8] That does not mean that the δ bound is of no interest; see §7.5.

used by several successful incremental algorithms. In Frederickson's algorithm [Fre85] and in the similar methods of Italiano [Ita91], large clusters of vertices in the graph are condensed into clusters, and links are maintained between the clusters. In Carroll and Ryder's algorithms for updating attribute and dominator graphs [CR88], a tree is laid over the target graph, and pointers can be followed along the tree edges, which are not in general the same as edges in the graph. Thus, lower bounds in the local persistence model, while interesting, really pertain to the class of incremental algorithms without data structures beyond those required to maintain the underlying graph. One might think of a bound in the local persistence model as stating: "This is the best you can do without complicated data structures."

7.5 A Direct Comparison of IRLB and δ-analysis

Examination of the results derived via IRLB analysis and δ-analysis yields certain situations in which the two models seem to give contradictory results. In this section we examine some of these results, which demonstrate that the two methods are essentially incomparable.

Consider, one more time, the problem of updating minimum spanning trees. Using δ-analysis, we demonstrated by Theorem 5.5.2 in Chapter 5 that this problem is non-δ-incremental in the local persistence problem. This means that any locally persistent algorithm might have to examine every vertex in the graph even for a very simple update. Using IRLB analysis, we found that minimum spanning tree has a Class 2 IRLB — an incremental algorithm cannot be better than 1 over the number of vertices faster than the optimal start-over algorithm. The Class 2 IRLB *does* allow for incremental algorithms that are faster than starting over, and in fact such an algorithm exists (Frederickson's algorithm [Fre85]). There is no contradiction here — even though very small updates might require looking at the entire graph (the δ-analysis result) *the incremental algorithm can still always beat starting over*. Furthermore, Frederickson's algorithm uses a complicated data structure that is not allowed in the local persistence model.

To take another example, updating shortest paths has a Class 3 IRLB, implying that

no incremental algorithm, in the worst case, can do better than looking at the whole graph. On the other hand, Ramalingam and Reps give a δ-incremental algorithm for updating shortest paths, specifically for the All Pairs Shortest Paths > 0 (APSP>0) problem [RR91]. However, there is no contradiction here because the worst case complexity of their algorithm can be as bad as starting over. The algorithm for APSP>0 has complexity $O(\|\delta\|_2 + \|\delta\|_1 \log \|\delta\|_1)$. Recall from §5.2 that the notation $\|\delta\|_2$ is the extended size of order 2 of δ, that is, δ, the neighbors of δ, and the neighbors of the neighbors of δ. In order to convert this figure into a complexity as a function of n, we need to look at the definition of $\|\delta\|$ used by Ramalingam and Reps.

Because of the nature of the APSP>0 problem, a vector of size n is maintained at each vertex. A similar issue to that discussed in §6.6.3 arises — how to count AFFECTED when the label at each vertex consists of a set or vector rather than a single discrete element. In Ramalingam and Reps, the definition of $\|\text{AFFECTED}\|$ is based on the sum of the $\|\text{AFFECTED}\|$ for the single-sink problem, summed over all vertices. Since AFFECTED can be as large as n, the $\|\text{AFFECTED}\|$ for the single-sink problem can be n^2; and $\|\text{AFFECTED}\|_2$, the relevant parameter for edge deletions, can be n^3. Summing over all vertices, we find that $\|\text{AFFECTED}\|_2$ for APSP>0 can be as large as n^4. Since the start over problem can be solved in $o(n^3)$, this certainly does not contradict the IRLB result. Note that a more careful, direct analysis of the complexity of the algorithm, which is not provided by Ramalingam and Reps, might yield a better bound. However, since the algorithm is based directly on Dijkstra's algorithm [Dij59] for shortest paths, it is certainly not going to be any better than $O(n^3)$.

The point of the above analysis is not to question the utility of the Ramalingam and Reps algorithm — when δ is small and the average degree of the graph is small, the algorithm may be a substantial improvement compared to starting from scratch. The point is that IRLB and δ-analysis look at the problem from different angles, and hence get different results. The issue is not which is "better"; rather, the information derived from the two methods may give a better understanding of the difficulty of finding incremental algorithms for certain problems. Thus we see the methods as complementary.

7.6 Another Approach – Complete Dynamic Problems

Reif has proposed another approach to analyzing incremental algorithms [Rei87]. By using Turing Machine reductions, he is able to categorize a group of problems as equally difficult to solve incrementally. While this does not yield a lower bound result, it does give some insight into the difficulty of developing incremental algorithms for a class of problems.

Reif begins by pointing out that there are a number of problems with linear time algorithms for which it appears to be difficult to develop an incremental algorithm. A list of such problems which are complete in deterministic polynomial time with respect to log space Turing Machine reductions includes:

1. acceptance of a linear time Turing Machine,
2. path system problems,
3. Boolean circuit evaluation,
4. unit resolution, and
5. depth-first search numbering of a graph

For all problems in the list, there exist linear time reductions in the sequential Random Access Machine model of computation. According to Reif, these problems also have the property that under a suitable encoding, the reductions themselves are *constant-time updatable*; that is, after a single change is made to the input for one problem, the reduction can be recomputed in constant time. This implies that if a sublinear algorithm can be found for updating one of these problems, it can be applied to update any of them.

Chapter 8

Conclusion

In this section, we summarize our results, and propose some open problems worthy of consideration.

8.1 Summary

Our first major results were in Chapter 3, in which we developed a theoretical framework for analyzing the complexity of incremental algorithms. Using this framework, we gave a general theorem for incremental algorithms (Theorem 3.2.3) which provides a relative lower bound for nearly all incremental algorithms, including those for *NP*-complete problems (Corollary 3.4.2). We also considered the precise conditions under which this theorem holds, and applied it to amortized complexity (Corollary 3.3.1).

In Chapter 4, we extended this framework by developing a general technique which yields tighter bounds on incremental algorithms for many problems. We presented the framework of a procedure that, given certain conditions, can yield an Incremental Relative Lower Bound (IRLB) (Theorem 4.2.4). For some problems, it is shown that a single update can take as long, in the worst case, as starting over: these are called Class 3 IRLB's. For other problems, which can be modeled as graphs, the worst case time for a single update can be no better than 1 over the number of vertices in the graph; since the number of vertices can be as small as the square root of the input size (vertices plus edges) this is a tighter bound than the bounds in Chapter 3. These bounds are called Class 2 IRLB's. This method can be used to provide new lower bounds for a number of incremental problems, such as: transitive closure, planarity testing, and strong connectivity (Class 3), and connected components, biconnected components, and minimum spanning tree (Class 2). We also discussed a class of problems based on

solving systems of equations and proved Class 3 and Class 2 lower bounds for these problems. The systems of equations can be used to model many problems in data flow analysis and other areas.

We examined, in Chapter 5, an alternative approach to incremental algorithms, δ-analysis, which was originally developed by others. Using this technique, we proved new incremental lower bounds for the connectivity, biconnectivity, and minimum spanning tree problems within the δ-analysis framework.

The problem of incremental transitive closure was addressed in Chapter 6. First, we used IRLB analysis to derive lower bounds for updating transitive closure in an arbitrary digraph (Class 3) and in an acyclic digraph (Class 2). Then using the δ-analysis model we developed new incremental algorithms for single-source reachability and transitive closure in an acyclic digraph. These new algorithms have the characteristic that they are δ-incremental; that is, the time to perform the update depends on the size of the region of the graph that needs to be updated rather than on the entire input graph. Thus these algorithms may be more efficient than other approaches for domains in which such regions are small relative to the size of the entire graph.

Finally, in Chapter 7 we analyzed the available approaches for understanding incremental algorithms. After considering two basic "flavors" of incremental problems (update versus online), we considered the relevant factors in choosing an approach to a particular incremental problem. The strengths and weaknesses of IRLB and δ-analysis were addressed. Since it can be shown that the two approaches are incomparable, we discuss how the apparently conflicting information can be reconciled to give a better understanding of incremental problems.

8.2 Future Work and Open Problems

The field of incremental algorithms has many open problems of both theoretical and practical interest. Based on our work, here are some of those that we believe would be most interesting for future research.

Problems related to IRLB's

1. Are there reasonable problems that do not have a fast initialization (Definition 3.2.1)? Can general conditions for the existence of a fast initialization be found?

2. IRLB proofs are based on algorithms that have a fixed limit on the amount of history information maintained. This limits, in effect, the amount of initialization or preprocessing that can be done by an algorithm covered by the IRLB result. Can this limitation be relaxed? Is there a trade-off between history information and speed of updating, or conversely, can it be shown that a large amount of history will slow down an incremental update?

3. Can other "models" of IRLB proofs be found, to apply to other types of problems? Can strong (Class 3) results be found for problems that are cannot be naturally modeled as graph problems? Is it possible to find Class 3 results for the types of problems that are shown to be in Class 2 — generally, problems modeled as undirected graphs?

4. Can good (Class 2 or Class 3) IRLB proofs be found that apply to the "easy" update (typically, additions)?

5. A *hypergraph* is like an undirected graph, except that edges connect arbitrary subsets of the vertices rather than pairs of vertices [CLR91, pp. 89–90]. Would problems on hypergraphs yield IRLB's falling between Class 2 and Class 3? There is also an issue here as to the "atomic" nature of a change when a single edge is a set of size $O(n)$.

Problems related to δ-analysis

1. Is there a better way to look at the extended size of δ? The current method does not generalize well to problems where the label at a node is a vector or set (see §6.6.3 and [RR91]).

2. Can δ-analysis be applied to domains other than graph theoretical problems?

3. Can a meta-proof technique, similar to the IRLB approach, be developed that would simplify the process of finding proofs of non-δ-incrementality? Most of the proofs have very similar structure.

4. Can a stronger model than local persistence be used to prove δ lower bounds?

Problems related to the Transitive Closure

1. Can an incremental update algorithm be found for TC-DAG that is efficient in a worst-case sense, i.e., meets the Class 2 IRLB or even is $o(n^3)$?

2. Can the algorithm given in §6.6 be generalized to apply to a larger class of graphs, e.g., reducible flow graphs [ASU86]?

3. Can a δ-incremental algorithm for TC-DAG be found that does better than $\|\delta\|_2$, or conversely, can it be shown that this is a lower bound?

4. Will empirical testing show that the δ-incremental algorithm for TC-DAG is practical, for random data? For data collected from a realistic domain?

General issues for incremental algorithms

1. Can a general model of incremental algorithms be developed that encompasses the different notions about update algorithms, online algorithms, and so forth, in a unified way?

2. Can better techniques be developed for getting stronger lower bounds results in the worst case? In the amortized case?

References

[ABJ89] R. Agrawal, A. Borgida, and H.V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 1989.

[ACR+87] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and F. K. Zadek. Incremental evaluation of attributed graphs. Technical Report RC 13205, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 1987.

[AHR+90] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42. Society for Industrial and Applied Mathematics, 1990.

[AHU74] A.Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Algorithms*. Addison–Wesley, 1974.

[AI91] G. Ausiello and G. F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *Journal of Logic Programming*, 10(1):69–90, 1991.

[AIMSN90] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 12–21. Society for Industrial and Applied Mathematics, 1990.

[AMSN89] G. Ausiello, A. Marchetti-Spaccamela, and U. Nanni. Dynamic maintenances of paths and path expressions in graphs. In *Proceedings of the International Symposium on Symbolic and Albegraic Computation*, pages 1–12, Berlin, 1989. Springer–Verlag. Lecture Notes in Computer Science 358.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986.

[BPR86] A. M. Berman, M. C. Paull, and B. G. Ryder. Proving relative lower bounds for incremental algorithms. Technical Report DCS-TR-154, Department of Computer Science, Rutgers University, 1986.

[BPR90] A. M. Berman, M. C. Paull, and B. G. Ryder. Proving relative lower bounds for incremental algorithms. *Acta Informatica*, 27:665–683, 1990.

[BR90] M. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7), July 1990.

[BT89] G. Di Battista and R. Tamassia. Incremental planarity testing. In *Proceedings of the Thirtieth Annual IEEE Symposium on the Foundations of Computer Science*, pages 436–441. Institute of Electrical and Electronics Engineers – Computer Society, 1989.

[BT90] G. Di Battista and R. Tamassia. On-line graph algorithms with spqr-trees. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming*. European Association for Theoretical Computer Science, 1990. Extended abstract.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Car88] M. Carroll. *Data flow analysis via attribute and dominator updates*. PhD thesis, Rutgers University, May 1988.

[CBT+92] R. F. Cohen, G. Di Battista, R. Tamassia, I. G. Tollis, and P. Bertolazzi. A framework for dynamic graph drawing. In *Proceedings of the Eighth Annual Symposium on Computational Geometry*. Association for Computing Machinery, 1992. Extended abstract.

[CC82] G. A. Cheston and D. G. Corneil. Graph property update algorithms and their application to distance matrices. *Infor*, 20(3):178–201, August 1982.

[CH78] F. Chin and D. Houck. Algorithms for updating minimum spanning trees. *Journal of Computer and System Sciences*, 16:333–344, 1978.

[Che76] G. A. Cheston. *Incremental Algorithms In Graph Theory*. PhD thesis, Univ. of Toronto, 1976. Department of Computer Science Tech. Report 91.

[Che84] G. A. Cheston. On-line connectivity algorithms. *Networks*, 14:83–94, 1984.

[CLR91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1991.

[CR88] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–284, January 1988.

[Cul91] P. Cull. Review of "Proving relative lower bounds for incremental algorithms" by A. M. Berman et al. *Computing Reviews*, page 375, July 1991.

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 1–6. Association for Computing Machinery, 1987.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[EG85] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods Oper. Research*, 49:371–387, 1985.

[EIT+92] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic planar graph. *Journal of Algorithms*, 13:33–54, 1992.

[ES81] S. Even and Y. Shiloach. An on-line edge deletion problem. *Journal of the ACM*, 28(1):1–4, January 1981.

[Fre85] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal of Computing*, 14(4):781–798, November 1985.

[FS84] G. N. Frederickson and M. A. Srinivas. On-line updating of degree-constrained minimum spanning trees. In *Proceedings of the Twenty-Second Allerton Conference on Communication, Control, and Computing*, October 1984.

[Fuj81] S. Fujishige. A note on the problem of updating shortest paths. *Networks*, 11:317–319, 1981.

[Gaz83] H. Gazit. Algorithms in dynamic graphs. Master's thesis, Technion, Haifa, Israel, May 1983. In Hebrew.

[GGST86] H. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[Gho83] V. Ghodssi. *Incremental analysis of programs.* PhD thesis, University of Central Florida, 1983.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman & Company, 1979.

[GSV78] S. Goto and A. Sangiovanni-Vincentelli. A new shortest path updating algorithm. *Networks*, 8:341–372, 1978.

[Har83] D. Harel. On line maintenance of the connected components of dynamic graphs. Unpublished manuscipt, 1983.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, 1977.

[HS76] E. Horowitz and S. Sahni. *Fundamentals of Data Structures.* Computer Science Press, 1976.

[IK83] T. Ibaraki and N. Katoh. On-line computation of transitive closure of graphs. *Information Processing Letters*, 16:95–97, 1983.

[Ita86] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoretical Computer Science*, 48:273–281, 1986.

[Ita88] G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Information Processing Letters*, 28:5–11, 1988.

[Ita91] G. F. Italiano. *Dynamic Data Structures for Graphs.* PhD thesis, Columbia University, 1991. Technical Report CUCS-019-91.

[Knu73] D. E. Knuth. *Fundamental Algorithms*, volume III of *The Art of Computer Programming.* Addison-Wesley, 1973.

[KRvM88] J. Keables, K. Roberson, and A. von Mayrhauser. Data flow analysis and its application to software maintenance. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 335–347. Institute of Electrical and Electronics Engineers – Computer Society, October 1988.

[LD91] H. R. Lewis and L. Denenberg. *Data Structures and Their Algorithms.* Harper Collins, 1991.

[LPv88] J. A. La Poutre and J. van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In H. Gottler and H. J. Schneider, editors, *Graph-Theoretic Concepts in Computer Science 1987*, volume 314 of *Lecture Notes in Computer Science*, pages 106–120, Berlin–Heidelberg–New York, 1988. Springer–Verlag.

[Mar89] T. J. Marlowe. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, October 1989. DCS-TR-225.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* EATCS Monographs on Theoretical Computer Science. Springer–Verlag, 1984.

[Ov81] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23(2):166–204, October 1981.

[PaCB84] M. C. Paull, C. Ching an Cheng, and A. M. Berman. Exploring the structure of incremental algorithms. Technical Report DCS-TR-139, Department of Computer Science, Rutgers University, May 1984. Preliminary version.

[Pau88] M.C. Paull. *Introduction to Algorithm Design Principles.* Wiley-Interscience, 1988.

[Pol86] L. L. Pollock. *An approach to incremental compilation of optimized code.* PhD thesis, University of Pittsburgh, April 1986.

[PR85] S. Pawagi and I. V. Ramakrishnan. Parallel updates of graph properties in logarithmic time. In *International Conference on Parallel Processing*, pages 186–193. Institute of Electrical and Electronics Engineers – Computer Society, 1985.

[PS89] L. L. Pollock and M. L. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12):1537–1549, December 1989.

[PT88] F. P. Preparata and R. Tamassia. Fully dynamic techniques for point location and transitive closure in planar structures. In *Proceedings of the Twenty-Ninth Annual IEEE Symposium on the Foundations of Computer Science*, pages 558–567. Institute of Electrical and Electronics Engineers – Computer Society, 1988.

[RC86] B. G. Ryder and M. D. Carroll. An incremental algorithm for software analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 171–179. Association for Computing Machinery, 1986.

[Rei87] J. H. Reif. A topological approach to dynamic graph connectivity. *Information Processing Letters*, 25:65–70, April 1987.

[Rep82] T. Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 169–176. Association for Computing Machinery, 1982.

[Rep88] T. Reps. Incremental evaluation for attribute grammars with unrestricted movement between tree modifications. *Acta Informatica*, 25:155–178, 1988.

[Rep92] T. Reps, March 1992. Personal Communication.

[RLP90] B. G. Ryder, W. Landi, and H. Pande. Profiling an incremental data flow analysis algorithm. *IEEE Transactions on Software Engineering*, 16(2):129–140, February 1990.

[Rod68] V. V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(5):336–343, 1968.

[RP86] B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[RP88] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[RR91] G. Ramalingam and T. Reps. On the computational complexity of incremental algorithms. Technical Report #1033, Computer Sciences Department, University of Wisconsin–Madison, 1991.

[RTD83] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[Ryd83] B. G. Ryder. Incremental data flow analysis. In *ACM Symposium on Principles of Programming Languages*, pages 167–176. Association for Computing Machinery, January 1983.

[Sac86] M. G. Sackrowitz. Incremental updating of depth-first search trees. Technical Report DCS-TR-173, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, January 1986.

[Sar86] N. Sarnak. *Persistent data structures.* PhD thesis, New York University, 1986.

[SP73] P. M. Spira and A. Pan. On finding and updating shortest paths and spanning trees. In *Symposium on Switching and Automata Theory*, pages 82–84. Institute of Electrical and Electronics Engineers – Computer Society, 1973.

[ST85] D. M. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 1985.

[Tam88] R. Tamassia. A dynamic data structure for planar graph embedding. In *Proceedings of the Fifteenth International Colloquium on Automata, Languages and Programming*, pages 576–590, Berlin, 1988. European Association for Theoretical Computer Science, Springer–Verlag. Lecture Notes in Computer Science 317.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, 1983.

[Tar84] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6(2):306–318, April 1984.

[TP90] R. Tamassia and F. P. Preparata. Dynamic maintenance of planar digraphs. *Algorithmica*, 5:509–527, 1990.

[Wes89] J. Westbrook. *Algorithms and data structures for dynamic graph problems.* PhD thesis, Princeton University, October 1989.

[WT92] J. Westbrook and R. E. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7:433–464, 1992.

[Yel91] D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. To be published in *Acta Informatica*, 1991.

[YS88] D. M. Yellin and R. Strom. INC: A language for incremental computation. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 115–124. Association for Computing Machinery, 1988.

[Zad84] F. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the SIGPLAN 1984 Conference on Programming Language Design and Implementation.* Association for Computing Machinery, July 1984.

Vita

Arthur Michael Berman

1974 Graduated from Palisades High School, Los Angeles, California.

1974–78 Attended Pomona College, Claremont, California. Majored in Economics.

1978 B.A., Pomona College.

1978–79 Research Assistant, Board of Governors of the Federal Reserve System, Washington, D.C.

1979–80 Applications Consultant, Tymshare Inc., Newark, New Jersey.

1980–92 Graduate work in Computer Science, Rutgers, The State University of New Jersey, New Brunswick, New Jersey.

1981–83 Teaching assistant/instructor, Rutgers, The State University of New Jersey.

1983 M.S. in Computer Science, Rutgers, The State University of New Jersey.

1984 Instructor, Computer Science, Duke University, Durham, North Carolina.

1986–87 Teaching assistant, Rutgers, The State University of New Jersey.

1988 Consultant, Siemens Research Laboratory, Princeton, New Jersey.

1989–92 Assistant Professor, Computer Science, Glassboro State College, Glassboro, New Jersey.

1990 A.M. Berman, M.C. Paull, and B.G. Ryder. Proving Relative Lower Bounds for Incremental Algorithms. *Acta Informatica*, 25:665–683.

1992 Ph.D. in Computer Science.