An Incremental Algorithm for a Generalization of the Shortest-Path Problem

G. Ramalingam and Thomas Reps
University of Wisconsin – Madison

The *grammar problem*, a generalization of the single-source shortest-path problem introduced by Knuth, is to compute the minimum-cost derivation of a terminal string from one or more non-terminals of a given context-free grammar, with the cost of a derivation being suitably defined. In this paper we present an incremental algorithm for a version of the grammar problem. As a special case of this algorithm we obtain an efficient incremental algorithm for the single-source shortest-path problem with positive edge lengths. The aspect of our incremental algorithm that distinguishes it from all other work on the dynamic shortest-path problem is its ability to handle "multiple heterogeneous modifications": between updates, the input graph is allowed to be restructured by an arbitrary mixture of edge insertions, edge deletions, and edge-length changes.

Categories and Subject Descriptors: E.1 [**Data Structures**] -- *graphs*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems -- *computations on discrete structures*; G.2.1 [**Discrete Mathematics**]: Combinatorics -- *combinatorial algorithms*; G.2.2 [**Discrete Mathematics**]: Graph Theory -- *graph algorithms*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: shortest-path problem, least-cost path problem, incremental algorithm, dynamic algorithm, Dijkstra's algorithm

---

1. Introduction

Knuth defined the following generalization of the single-source shortest-path problem, called the *grammar problem* [17]: Consider a context-free grammar in which every production is associated with a real-valued function whose arity equals the number of non-terminal occurrences on the right-hand side of the production. Every derivation of a terminal string from a non-terminal has an associated derivation tree; replacing every production in the derivation tree by the function associated with that production yields an expression tree. Define the cost of a derivation to be the value of the expression tree obtained from the derivation. The goal of the grammar problem is to compute the minimum-cost derivation of a terminal string from one or more non-terminals of the given grammar.

Knuth showed that it is possible to adapt Dijkstra's shortest-path algorithm [6] to solve the grammar problem if the functions defining the costs of derivations satisfy a simple property (see Section 2). In addition to the single-source shortest-path problem, Knuth lists a variety of other applications and special cases of the grammar problem, including the generation of optimal code for expression trees and the construction of optimal binary-search trees.

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

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

In this paper, we present an algorithm for a version of the *dynamic grammar problem*—the problem of updating the solution to the grammar problem when the input grammar is modified. As a special case of the algorithm, we obtain a new, simple, and efficient algorithm for the dynamic single-source shortest-path problem with positive edge lengths (the *dynamic SSSP>0 problem*).

The aspect of our work that distinguishes it from all other work on dynamic shortest-path problems is that the algorithm we present handles *multiple heterogeneous changes*: Between updates, the input graph is allowed to be restructured by an arbitrary mixture of edge insertions, edge deletions, and edge-length changes.[1] Most previous work on dynamic shortest-path problems has addressed the problem of updating the solution after the input graph undergoes either *unit changes—i.e*, exactly one edge is inserted, deleted, or changed in length—or else *homogeneous changes—i.e.*, changes to multiple edges are permitted, but all changes must be of the same kind: either all insertions/length-decreases or all deletions/length-increases. (A comprehensive comparison of our work with previous work appears in Section 7.)

In general, a single application of an algorithm for heterogeneous changes has the potential to perform significantly better than either the repeated application of an algorithm for unit changes or the double application of an algorithm for homogeneous changes. There are two sources of potential savings: *combining* and *cancellation*.

Combining
: If updating is carried out by using multiple applications of an algorithm for unit or homogeneous changes, a vertex might be examined several times, with the vertex being assigned a new (but temporary and non-final) value on each visit until the last one. An algorithm for heterogeneous changes has the potential to combine the effects of all of the different modifications to the input graph, thereby eliminating the extra vertex examinations.

Cancellation
: The effects of insertions and deletions can cancel each other out. Thus, if updating is carried out by using multiple applications of an algorithm for unit or homogeneous changes, superfluous work can be performed. In one updating pass, vertices can be given new values only to have a subsequent updating pass revisit the vertices, restoring their original values. With an algorithm for heterogeneous changes, there is the potential to avoid such needless work.

The updating algorithm presented in this paper exploits these sources of potential savings to an essentially optimal degree: if the initial value of a vertex is already its correct, final value, then the value of that vertex is never changed during the updating; if the initial value of a vertex is incorrect, then either the value of the vertex is changed only once, when it is assigned its correct final value, or the value of the vertex is changed exactly twice, once when the value is temporarily changed to ∞, and once when it is assigned its correct, final value. (Bear in mind that, when updating begins, it is not known which vertices have correct values and which do not.)

The behavior of the algorithm is best characterized using the notion of a *bounded incremental algorithm*: An algorithm for a dynamic problem is said to be a bounded incremental algorithm if the time it takes to update the solution is bounded by some function of $\|\delta\|$, where $\|\delta\|$ is a measure of "the size of the change in the input and output". (For a formal definition of the notion of boundedness, see Section 2.) In the case of the dynamic SSSP>0 problem, after an arbitrary mixture of edge insertions, edge deletions, and

[1]The operations of inserting an edge and decreasing an edge length are equivalent in the following sense: The insertion of an edge can be considered as the special case of an edge length being decreased from ∞ to a finite value, while the case of a decrease in an edge length can be considered as the insertion of a new edge parallel to the relevant edge. The operations of deleting an edge and increasing an edge length are similarly related.

edge-length changes, the algorithm presented in Section 3 updates a graph in time $O(\|\delta\| \log \|\delta\|)$.[2]

Dijkstra's algorithm turns out to be a special case of our algorithm for the dynamic SSSP>0 problem: when a collection of edges is inserted into an empty graph, our algorithm works like Dijkstra's algorithm. Similarly, a variant of Knuth's algorithm for the batch grammar problem is obtained as a special case of our algorithm for the dynamic grammar problem. Note, however, that our incremental algorithms encounter "configurations" that can never occur in any run of the batch algorithms. For example, in the dynamic SSSP>0 algorithm, a vertex u can, at some stage, have a distance $d(u)$ that is *strictly less than* $\min_{v \in Pred(u)} [d(v) + length(v \rightarrow u)]$. This situation never occurs in Dijkstra's algorithm.

In the paper, we describe an application of the incremental algorithm for the dynamic SSSP>0 problem to batch shortest-path problems on graphs that have a small number of negative edges (but no negative-length cycles). Yap proposed an algorithm for finding the shortest path between two vertices (*i.e.*, for the single-pair shortest-path problem) in graphs with a few negative edges [28]. Yap's algorithm is more efficient than the standard algorithms that handle an arbitrary number of negative edges; however, the incremental SSSP>0 algorithm can be employed in an algorithm that is even more efficient. In addition, unlike Yap's algorithm—which is restricted to the single-*pair* problem in graphs with a few negative edges—our algorithm can also be used to solve the single-*source* problem in graphs with a few negative edges.

This paper is organized as follows. In Section 2 we define the problem to be solved and introduce the terminology we use. In Section 3 we develop the idea behind the algorithm via a sequence of lemmas about the problem. We present the first version of our algorithm, a proof of its correctness, and an analysis of its time complexity in Section 4. In Section 5, we discuss an improved version of the first algorithm, and analyze its time complexity. In Section 6 we look at some extensions of the algorithm. In Section 7 we discuss related work. The paper ends with an appendix that covers some relevant results and their proofs.

---

[2]The concept of boundedness is interesting because it enables us to distinguish between various incremental algorithms and batch algorithms. For instance, when the cost of the computation is expressed as a function of the size of the (current) input, all incremental algorithms that have been proposed for updating the solution to the (various versions of the) shortest-path problem after the deletion of a single edge run in time asymptotically no better, in the worst-case, than the time required to perform the computation from scratch. Spira and Pan [26], in fact, show that no incremental algorithm for this problem can do better than the best batch algorithm, under the assumption that the incremental algorithm retains only the shortest-paths information. In other words, with the usual way of analyzing incremental algorithms—worst-case analysis in terms of the size of the current input—no incremental shortest-path algorithm would appear to be any better than merely employing the best batch algorithm to recompute shortest paths from scratch! In contrast, the incremental algorithm for SSSP>0 presented in this paper is bounded and runs in time $O(\|\delta\| \log \|\delta\|)$, whereas any batch algorithm for SSSP>0 will be an unbounded incremental algorithm.

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

> An algorithm with bad worst-case complexity will have good amortized complexity only if there is something about the problem being updated, or about the way in which we update it, or about the kinds of updates which we allow, that precludes pathological updates from happening frequently [3].

Thus, Ausiello *et al.* [1], for instance, use amortized-cost analysis to obtain a better bound on the time complexity of an algorithm they present for maintaining shortest paths in a graph as the graph undergoes a sequence of edge insertions. However, in the fully dynamic version of the shortest-path problem, where both edge insertions and edge deletions are allowed, "pathological" input changes can occur frequently in a sequence of input changes. Thus, when costs are expressed as a function of the size of the input, the amortized-cost complexity of algorithms for the fully dynamic version of the shortest-path problem will not, in general, be better than their worst-case complexity.

2. Terminology, Notation and the Definition of the Problem

A *directed graph* $G = (V(G), E(G))$ consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$. We denote an edge directed from u to v by $u \rightarrow v$. If $u \rightarrow v$ is an edge in the graph, we say that u is the *source* of the edge, that v is the *target*, that u is a *predecessor* of v, and that v is a *successor* of u. The set of all predecessors of a vertex u will be denoted by $Pred(u)$. If U is a set of vertices, then $Pred(U)$ is defined to be $\bigcup_{u \in U} Pred(u)$. The sets $Succ(u)$ and $Succ(U)$ are defined similarly.

The Shortest-Path Problem

The input for the various versions of the shortest-path problem typically consists of a directed graph in which every edge $u \rightarrow v$ has an associated real-valued *length*, which we denote by $length(u \rightarrow v)$. The length of a path in the graph is defined to be the sum of the lengths of the edges in the path, each edge contributing to the length as many times as it occurs in the path.

The ***single-source shortest-path*** problem, abbreviated SSSP, is the problem of determining for every vertex u in a given graph G (the length of) a shortest path from a distinguished *source* vertex of the given graph, denoted by $source(G)$, to u. (In this paper, we concentrate on the problem of computing the length of shortest paths, rather than that of finding the shortest paths themselves. We show later, in Section 6.1, that our algorithm can be easily extended to the problem of finding the shortest paths too.) Two simpler versions of this problem are obtained by restricting the input instances to be graphs in which every edge has a non-negative or positive length respectively. We refer to these two problems as the SSSP≥0 problem and SSSP>0 problem, respectively.

Every input instance G of SSSP induces a collection of equations, the *Bellman-Ford* equations, in the set of unknowns $\{ d(u) \mid u \in V(G) \}$:

$$\begin{aligned} d(u) &= 0 && \text{if } u = source(G) \\ &= \min_{v \in Pred(u)} [d(v) + length(v \rightarrow u)] && \text{otherwise} \end{aligned}$$

It can be shown that the maximal fixed point of this collection of equations is the solution to the SSSP problem if the input graph contains no negative length cycles. (See [13] for instance.) It is necessary to view the unknowns as belonging to the set of reals extended by $+\infty$ so that for every vertex u unreachable from the source vertex $d(u)$ will be ∞ in the maximal fixed point, as required. Further, if all edge lengths are positive, then the above collection of equations has a unique fixed point. Hence, the SSSP>0 problem may be viewed as the problem of solving the above collection of equations.

The Grammar Problem

Let us now consider a generalization of the shortest path problem due to Knuth. In the rest of the paper let $(D, \leq, \infty)$ be a totally ordered set with maximum element ∞.[3] We define an ***abstract grammar*** (with a value domain of D) to be a context-free grammar in which all productions are of the general form

$$Y \rightarrow g(X_1, \ldots, X_k),$$

where $Y, X_1, \ldots, X_k$ are non-terminal symbols, and g, the parentheses, and the commas are all terminal symbols. In addition, each production $Y \rightarrow g(X_1, \ldots, X_k)$ has an associated function from D^k to D, which will be denoted by g itself in order to avoid the introduction of more notation. We will refer to the

[3] Knuth uses the specific totally ordered set $(\mathcal{R}_+, \leq, \infty)$, where $\mathcal{R}_+$ denotes the set of non-negative reals extended with the value ∞, and $\leq$ is the usual ordering on reals.

function g as a ***production function*** of the given grammar.

For every non-terminal symbol Y of an abstract grammar G over the terminal alphabet T we let $L_G(Y) = \{ \alpha \mid \alpha \in T^* \text{ and } Y \rightarrow^* \alpha \}$ be the set of terminal strings derivable from Y. Every string α in $L(Y)$ denotes a composition of production functions, so it corresponds to a uniquely defined value in D, which we shall call ***val(*α*)***. The ***grammar problem*** is to compute the value $m_G(Y)$ for each non-terminal Y of a given abstract grammar G, where

$$m_G(Y) =_{def} \min \{ val(\alpha) \mid \alpha \in L_G(Y) \}.$$

Note that in general the set $\{ val(\alpha) \mid \alpha \in L_G(Y) \}$ need not have a minimum element, and, hence, $m_G(Y)$ need not be well defined. However, some simple restrictions on the type of production functions allowed, which we discuss soon, guarantee that $m_G(Y)$ is well defined.

We now consider some applications and special cases of the grammar problem given in [17]. Given a context-free grammar, consider the abstract grammar obtained by replacing each production $Y \longrightarrow \theta$ in the given grammar by the production $Y \longrightarrow g_\theta(X_1, \ldots, X_k)$, where $X_1, \ldots, X_k$ are the non-terminal symbols occurring in θ from left to right (including repetitions). If we define the production function g_θ by

$$g_\theta(x_1, \ldots, x_k) =_{def} x_1 + \cdots + x_k + (\textit{the number of terminal symbols in } \theta)$$

then $m_G(Y)$, the solution to the resulting grammar problem, is the length of the shortest terminal string derivable from non-terminal Y. If we instead define g_θ by

$$g_\theta(x_1, \ldots, x_k) =_{def} max(x_1, \ldots, x_k) + 1$$

then $m_G(Y)$ is the minimum height of a parse tree for a string derivable from the non-terminal Y.

Let us now see how the grammar problem generalizes the single-source shortest-path problem. Every input instance of the single-source shortest-path problem can be easily transformed into an input instance of the grammar problem whose solution yields the solution to the original problem as follows. The new grammar consists of one non-terminal N_u for every vertex u in the given graph. For every edge $u \rightarrow v$ in the graph, we add a new terminal $g_{u \rightarrow v}$, and the production $N_v \longrightarrow g_{u \rightarrow v}(N_u)$, where the production function corresponding to $g_{u \rightarrow v}$ is given by $g_{u \rightarrow v}(x) = x + length(u \rightarrow v)$. In addition, we add the production $N_s \longrightarrow 0$, where s is the source vertex, and 0 is a terminal representing the constant-valued function zero.

Thus, the single-source shortest-path problem (with non-negative edge lengths) corresponds to the special case of the grammar problem where the input grammar is regular, and all the production functions g are of the form $g(x) = x + h$ (for some $h \geq 0$) or $g() = 0$. (Strictly speaking, the grammar encoding a shortest-path problem is not a regular grammar because of the use of the parentheses. However, this is immaterial since the parentheses were used in the definition of the grammar problem just as a notational convenience.) Further, the grammar problem corresponds to an SSSP problem only if contains exactly one production of the form $N \longrightarrow 0$; if more than one production is of this form, then we have a "simultaneous multi-source shortest-path problem".

Knuth shows how Dijsktra's algorithm for computing shortest paths can be generalized to solve the grammar problem for a special class of abstract grammars, namely the class of SF grammars, which is defined as follows.

A function $g(x_1, \ldots, x_k)$ from D^k to D is said to be a ***superior function*** (abbreviated *s.f.*) if it is monotone non-decreasing in each variable and if $g(x_1, \ldots, x_k) \geq max(x_1, \ldots, x_k)$. A function $g(x_1, \ldots, x_k)$ from D^k to D is said to be a ***strict superior function*** (abbreviated *s.s.f.*) if it is monotone non-decreasing in each variable and if $g(x_1, \ldots, x_k) > max(x_1, \ldots, x_k)$. An abstract grammar in which every production function is a superior function is said to be an ***SF grammar***. An abstract grammar in which every production function is a strict superior function is said to be an ***SSF grammar***. Examples of superior functions

over $(\mathcal{R}_+, \leq, \infty)$ include $max(x_1, \ldots, x_k)$, $x+y$, and $\sqrt{x^2+y^2}$. None of these functions are strict superior functions over the set of non-negative reals, although the later two are strict superior functions over the set of positive reals.

Note that the abstract grammar generated by an instance of the SSSP≥0 problem is an SF grammar, while the abstract grammar generated by an instance of the SSSP>0 problem is an SSF grammar.

Every instance of the grammar problem, much like every instance of the shortest-path problem, determines a collection of mutually recursive equations, which consists of the following equation for each non-terminal Y in the grammar:

$$d(Y) = \min \{ g(d(X_1), \ldots, d(X_k)) \mid Y \longrightarrow g(X_1, \ldots, X_k) \text{ is a production } \}. \qquad (*)$$

It can be shown that the above collection of equations has a maximal fixed point if the given grammar is an SF grammar, and that this maximal fixed point yields the solution to the SF grammar problem. Further, as was shown by Knuth, the above collection of equations has a unique fixed point if the given grammar is an SSF grammar. Some of these results are established in the appendix of this paper.

The Fixed Point Problem

Our approach to these problems is to view them as the computation of the unique fixed point of a collection of equations satisfying certain properties. One of the advantages of this approach is in proving the correctness of an algorithm for these problems: all we need to show is that the set of values computed by the algorithm satisfy a particular set of equations.

We now define two classes of functions that generalize the class of superior and strict superior functions respectively, which will be utilized in defining the fixed point problem. Let $[i,k]$ denote the set of integers $\{ j \mid i \leq j \leq k \}$. We say a function $g : D^k \longrightarrow D$ is a ***weakly superior function*** (abbreviated *w.s.f.*) if it is monotone non-decreasing in each variable and if for every $i \in [1,k]$,

$$g(x_1, \ldots, x_i, \ldots, x_k) < x_i \Rightarrow g(x_1, \ldots, x_i, \ldots, x_k) = g(x_1, \ldots, \infty, \ldots, x_k).$$

We say a function $g : D^k \longrightarrow D$ is a ***strict weakly superior function*** (abbreviated *s.w.s.f.*) if it is monotone non-decreasing in each variable and if for every $i \in [1,k]$,

$$g(x_1, \ldots, x_i, \ldots, x_k) \leq x_i \Rightarrow g(x_1, \ldots, x_i, \ldots, x_k) = g(x_1, \ldots, \infty, \ldots, x_k).$$

It can be easily verified that every *s.f.* is also a *w.s.f.*, while every *s.s.f.* is also an *s.w.s.f.* The function $min(x_1, \ldots, x_k)$ is an example of a *w.s.f.* that is not an *s.f.*, while $min(x_1, \ldots, x_k)+1$ is an example of an *s.w.s.f.* that is not an *s.s.f.* A constant-valued function is another example of an *s.w.s.f.*

Now consider a collection Q of k equations in the k unknowns x_1 through x_k, the i-th equation being

$$x_i = g_i(x_1, \ldots, x_k). \qquad (\dagger)$$

An equation of the above form is said to be a WSF equation if g_i is a *w.s.f.*, and an SWSF equation if g_i is an *s.w.s.f.*

It can be shown that if each of the equations in Q is a WSF equation then Q has a maximal fixed point. We define the ***WSF maximal fixed point problem*** to be that of computing the maximal fixed point of a collection of WSF equations. This problem generalizes the SF grammar problem, since, as we show in the appendix, the equation (*) determined by an SF grammar is a WSF equation. An adaptation of Dijkstra's algorithm can, in fact, be used to solve the WSF equation problem.

However, we are interested in incremental algorithms for the fixed point problem, and it turns out to be necessary to address a restricted version of the WSF equation problem. If each of the equations in Q is an

SWSF equation, then Q can be shown to have a unique fixed point. We define the ***SWSF fixed point problem*** to be that of computing the unique fixed point of a collection of SWSF equations. The SWSF fixed point problem generalizes the SSF grammar problem, since each equation in the collection of equations determined by an SSF grammar is an SWSF equation, as we show in the appendix. (The SSSP>0 problem is obtained as yet a further special case of the SSF grammar problem; that is, when all edge lengths are positive, the Bellman-Ford equations are all SWSF.)

Note that the expression on the right-hand side of the i-th equation (see (†)) need not contain all the variables and that the i-th equation may be more precisely written as

$$x_i = g_i(x_{j_{i,1}}, x_{j_{i,2}}, \ldots, x_{j_{i,n(i)}}).$$

We will continue to use the earlier form of the equation as a notational convenience although an algorithm to compute the fixed point of the collection of equations can use the *sparsity* of the equations to its advantage. We define the ***dependence graph*** of the collection Q of equations to be the graph (V,E) where $V = \{ x_i \mid 1 \le i \le k \}$, and $E = \{ x_j \rightarrow x_i \mid x_j$ occurs in the right-hand-side expression of the equation for $x_i \}$. For the sake of brevity we will often not distinguish between the collection of equations and the corresponding dependence graph. For instance, we will refer to the variable x_i as "vertex x_i".

For convenience, we will refer to the function associated with a vertex x_i by both g_i and g_{x_i}. Since the function g_i is part of the input, g_i is also referred to as the input value associated with vertex x_i. The value that the unknown x_i has in the (maximal) fixed point of the given collection of equations is referred to as the output value associated with vertex x_i.

Boundedness

Consider an input instance G of the SWSF fixed point problem. An *input modification* δ to G may change the equation associated with one or more of the vertices in G (simultaneously inserting or deleting edges from G). We denote the resulting graph by $G+\delta$. A vertex u of $G+\delta$ is said to be a ***modified*** vertex if the equation associated with u was changed. The set of all modified vertices in $G+\delta$ will be denoted by $Modified_{G,\delta}$. This set captures the change in the input.

A vertex in $G+\delta$ is said to be an ***affected*** vertex if its output value in $G+\delta$ is different from its output value in G. Let $Affected_{G,\delta}$ denote the set of all affected vertices in $G+\delta$. This set captures the change in the output. We define $Changed_{G,\delta}$ to be $Modified_{G,\delta} \cup Affected_{G,\delta}$. Thus, $Changed_{G,\delta}$ captures the change in the input *and* output.

A key aspect of the analysis of our algorithm is that we will be expressing the complexity of the algorithm in terms of the "size of the change in input and output".

The cardinality of a set of vertices K in a graph G will be denoted by $|K|$. For our purposes, a more useful measure of the "size" of K is the ***extended size*** $\|K\|_G$ of K, which is defined to be the sum of the number of vertices in K and the number of edges which have at least one endpoint in K.

Thus, the two parameters we will find useful are $|Changed_{G,\delta}|$, which we abbreviate to $|\delta|_G$, and $\|Changed_{G,\delta}\|_{G+\delta}$, which we abbreviate to $\|\delta\|_G$. The subscripts G and δ in the above terms will be omitted if no confusion is likely.

An incremental algorithm for the SWSF problem is said to be ***bounded*** if we can bound the time taken for the update step by a function of the parameter $\|\delta\|_G$ (as opposed to other parameters, such as $|V(G)|$

or $|G|$).[4] It is said to be ***unbounded*** if its running time can be arbitrarily large for fixed $\|\delta\|_G$. Thus, a bounded incremental algorithm is an incremental algorithm that processes only the "region" where the input or the output changes.

While the above definition of boundedness is applicable for the shortest-path problem, it needs to be generalized for the SWSF problem since the cost of updating the solution to the SWSF problem after a change in the input will depend on the cost of computing the various functions associated with the vertices. The following definition is motivated by the observation that an incremental algorithm that processes only the "region" where the input or the output changes will evaluate only the functions associated with vertices in $Changed \cup Succ(Changed)$. Define $C_{G,\delta}$ (abbreviated C_δ) to be the maximum over all vertices in $Changed \cup Succ(Changed)$ of the cost of evaluating the function associated with that vertex. An incremental algorithm for the SWSF problem is said to be a ***bounded scheduling cost algorithm*** if we can bound the time taken for the update step by a function of the parameters $\|\delta\|_G$ and $C_{G,\delta}$. The algorithms presented in Sections 4 and 5 are both bounded scheduling cost algorithms. The algorithm presented in Section 5 is a bounded incremental algorithm for the special case of the dynamic SSSP>0 problem.

3. The Idea Behind the Algorithm

We are given a dependence graph with n vertices $x_1, \ldots, x_n$. Every vertex x_i has an associated *s.w.s.f.* function g_i, and also an associated tentative output value $d(x_i)$, which denotes the correct output value of the vertex before the dependence graph was modified. Let $d^*(x_i)$ denote the actual output value that vertex x_i should have in the unique fixed point of the given collection of equations. Note that most of the following terminology is relative to a given assignment d. The ***rhs*** value of a vertex x_i, denoted by $rhs(x_i)$, is defined to be $g_i(d(x_1), \ldots, d(x_k))$. We say that vertex x_i is ***consistent*** if

$$d(x_i) = g_i(d(x_1), \ldots, d(x_k))$$

and that x_i is ***inconsistent*** otherwise. Two possible types of inconsistency can be identified. We say x_i is an ***over-consistent vertex*** if

$$d(x_i) > g_i(d(x_1), \ldots, d(x_k)).$$

We say x_i is an ***under-consistent vertex*** if

$$d(x_i) < g_i(d(x_1), \ldots, d(x_k)).$$

A vertex u is said to be a ***correct*** vertex if $d(u) = d^*(u)$, an ***over-estimated*** vertex if $d(u) > d^*(u)$, and an ***under-estimated*** vertex if $d(u) < d^*(u)$. Because $d^*(u)$ is not known for every vertex u during the updating, an algorithm can only make use of information about the "consistency status" of a given vertex, rather than its "correctness status". The notions of over-estimated and under-estimated vertices are introduced because they are needed for proving the correctness of the algorithm.

We have already seen that the SSSP>0 problem is a special case of the SWSF fixed point problem. Our incremental algorithm for the dynamic SWSF fixed point problem can best be explained as a generalization of Dijkstra's algorithm for the batch shortest-path problem. To draw out the analogy, let us summarize Dijkstra's algorithm using the above terminology.

The collection of equations to be solved in the case of the SSSP>0 problem is the collection of Bellman-Ford equations. In Dijkstra's algorithm all vertices initially have a value of ∞. At any stage of the algorithm, some of the vertices will be consistent while all the remaining vertices will be over-consistent. The

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

algorithm "processes" the inconsistencies in the graph in a particular order: at every stage, it chooses an over-consistent vertex x_i for which the rhs value is minimum, and "fixes" this inconsistency by changing $d(x_i)$ to $rhs(x_i)$. The algorithm derives its efficiency by processing the inconsistencies in the "right order", which guarantees that it has to process every vertex at most once.

The idea behind our algorithm is the same, namely to process the inconsistencies in the graph in the right order. The essential difference between our algorithm (for the fully dynamic problem) and Dijkstra's algorithm (for the static problem) is that we need to handle under-consistent vertices too. Under-consistent vertices can arise in the dynamic shortest-path problem, for instance, when some edge on some shortest path is deleted. We first establish via a sequence of lemmas that the inconsistencies in the graph should be processed in increasing order of ***key***, where the key of an inconsistent vertex x_i, denoted by $key(x_i)$, is defined as follows:

$$key(x_i) =_{def} min(\, d(x_i),\ g_i(d(x_1), \ldots, d(x_k)) \,).$$

In other words, the key of an over-consistent vertex x_i is $g_i(d(x_1), \ldots, d(x_k))$, while the key of an under-consistent vertex x_i is $d(x_i)$.

The following sequence of lemmas essentially establish two results concerning the order in which inconsistencies are to be processed. The first result addresses the concern that processing inconsistencies in an arbitrary order may entail "unnecessary" work and lead to an unbounded algorithm. An inconsistent vertex need not in general be incorrect; an under-consistent vertex need not in general be an under-estimated vertex; and an over-consistent vertex need not in general be an over-estimated vertex. (This is not true in the case of Dijkstra's algorithm, where under-consistent vertices cannot exist, and every overconsistent vertex is guaranteed to be an over-estimated vertex.) If we change the value of an inconsistent but correct vertex to make it consistent, we may end up with an unbounded algorithm. However, as we show below in Lemmas 3.9 and 3.10, if u is the inconsistent vertex with the least key, then u is guaranteed to be an over-estimated vertex if it is over-consistent, and it is guaranteed to be an under-estimated vertex if it is under-consistent.

The second result addresses the question of how an inconsistent vertex is to be processed. We show in Lemma 3.11 that if the inconsistent vertex with the least key is over-consistent, then its correct value can be immediately established. No such result holds true for under-consistent vertices; however, it turns out that an under-consistent vertex can be "processed" by simply setting its value to ∞, thereby converting it into either a consistent vertex or an over-consistent vertex.

We first establish some properties of *s.w.s.f.* functions that will be useful later on. Thinking about an *s.w.s.f.* of the form $min(x_1+h_1, \ldots, x_k+h_k)$, where each $h_i > 0$ may make it easier to understand the proposition.

Proposition 3.1.

(a) Let $g : D^k \longrightarrow D$ be a *s.w.s.f.* and let $I \subseteq \{1, \ldots, k\}$ be such that $g(x_1, \ldots, x_k) \leq x_i$ for every $i \in I$. Then,

$$g(y_1, \ldots, y_k) = g(x_1, \ldots, x_k)$$

where $y_i =_{def}$ if $(i \in I)$ then ∞ else x_i.

(b) Let $g : D^k \longrightarrow D$ be a *s.w.s.f.* and let $x_1, \ldots, x_k$ be such that $g(x_1, \ldots, x_i, \ldots, x_k) \leq x_i$. Then,

(1) $g(x_1, \ldots, y, \ldots, x_k) = g(x_1, \ldots, x_i, \ldots, x_k)$ for all $y \geq g(x_1, \ldots, x_i, \ldots, x_k)$.
(2) $g(x_1, \ldots, y, \ldots, x_k) > y$ for all $y < g(x_1, \ldots, x_i, \ldots, x_k)$.

(*c*) If g is a *s.w.s.f*. then

$$g(x_1, \ldots, x_k) < g(y_1, \ldots, y_k) \Rightarrow \exists i \in [1,k] \text{ such that } x_i < g(x_1, \ldots, x_k) \text{ and } x_i < y_i.$$

Proof.

(*a*) This follows by repeated applications of the definition of an *s.w.s.f*.

(*b*) Let $x_1, \ldots, x_k$ be such that $g(x_1, \ldots, x_i, \ldots, x_k) \leq x_i$. We now prove (1). Let $y \geq g(x_1, \ldots, x_i, \ldots, x_k)$. We show that $g(x_1, \ldots, y, \ldots, x_k) = g(x_1, \ldots, x_i, \ldots, x_k)$ by assuming otherwise and deriving a contradiction.

$$\begin{aligned}
&g(x_1, \ldots, y, \ldots, x_k) \neq g(x_1, \ldots, x_i, \ldots, x_k) \\
&\quad \Rightarrow g(x_1, \ldots, y, \ldots, x_k) \neq g(x_1, \ldots, \infty, \ldots, x_k) && \text{(since } g \text{ is strictly weakly superior)} \\
&\quad \Rightarrow g(x_1, \ldots, y, \ldots, x_k) < g(x_1, \ldots, \infty, \ldots, x_k) && \text{(since } g \text{ is monotonic)} \\
&\quad \Rightarrow g(x_1, \ldots, y, \ldots, x_k) < g(x_1, \ldots, x_i, \ldots, x_k) && \text{(since } g \text{ is strictly weakly superior)} \\
&\quad \Rightarrow g(x_1, \ldots, y, \ldots, x_k) < y && \text{(from assumption about } y\text{)} \\
&\quad \Rightarrow g(x_1, \ldots, y, \ldots, x_k) = g(x_1, \ldots, \infty, \ldots, x_k) && \text{(since } g \text{ is strictly weakly superior)} \\
&\quad \Rightarrow g(x_1, \ldots, y, \ldots, x_k) = g(x_1, \ldots, x_i, \ldots, x_k) && \text{(since } g \text{ is strictly weakly superior)}
\end{aligned}$$

The result follows. Now (2) follows as a simple consequence of (1). Suppose there exists some $y < g(x_1, \ldots, x_i, \ldots, x_k) \leq x_i$ such that $g(x_1, \ldots, y, \ldots, x_k) \leq y$. Thus, we have $g(x_1, \ldots, y, \ldots, x_k) \leq y$ and $x_i \geq g(x_1, \ldots, y, \ldots, x_k)$. Using (1), but with the roles of x_i and y reversed, we have $g(x_1, \ldots, x_i, \ldots, x_k) = g(x_1, \ldots, y, \ldots, x_k) \leq y$, which is a contradiction.

(*c*) We prove the contrapositive. Assume that the conclusion is false. Hence, for every $x_i < g(x_1, \ldots, x_k)$ we have $x_i \geq y_i$. Then,

$$\begin{aligned}
g(x_1, \ldots, x_k) &= g(z_1, \ldots, z_k) && \text{where } z_i =_{def} \text{ if } (x_i \geq g(x_1, \ldots, x_k)) \text{ then } \infty \text{ else } x_i \\
&&& \text{(from } (a)\text{)} \\
&\geq g(y_1, \ldots, y_k) && \text{since every } z_i \geq y_i. \\
&&& \text{(since } g \text{ is monotonic)}
\end{aligned}$$

The result follows. □

It is worth restating our goal at this point: we wish to show how the "correctness" status of some vertices can be inferred from the "consistency" status of various vertices. In particular, we want to show that u, the inconsistent vertex with the least key, must be an over-estimated vertex if it is over-consistent, and that u must be an under-estimated vertex if it is under-consistent. We do so by assuming that an over-consistent vertex u is not an over-estimated vertex and showing that there exists another inconsistent vertex v such that $key(v) < key(u)$, and by establishing a similar result for under-consistent vertices. We begin by showing that if an over-consistent vertex u is not an over-estimated vertex or if an under-consistent vertex u is not an under-estimated vertex then certain "local conditions" must hold true, *i.e.*, that the predecessors of vertex u must satisfy certain conditions.

Lemma 3.2. If u is any vertex such that $g_u(d(x_1), \ldots, d(x_k)) > d^*(u)$, then there exists an over-estimated predecessor v of u such that $d^*(v) < d^*(u)$.

Proof.

$$\begin{aligned}
&g_u(d(x_1), \ldots, d(x_k)) > d^*(u) \\
&\quad \Rightarrow g_u(d(x_1), \ldots, d(x_k)) > g_u(d^*(x_1), \ldots, d^*(x_k)) && \text{(by definition of } d^*(u)\text{)} \\
&\quad \Rightarrow d^*(x_i) < g_u(d^*(x_1), \ldots, d^*(x_k)) \text{ and } d^*(x_i) < d(x_i) \text{ for some } i && \text{(from Proposition 3.1)} \\
&\quad \Rightarrow d^*(x_i) < d^*(u) \text{ and } x_i \text{ is an overestimated predecessor of } u && \text{(by definition of } d^*(u)\text{)}
\end{aligned}$$

□

Lemma 3.3. If u is an under-consistent vertex, then u must be an under-estimated vertex or u must have an over-estimated predecessor v such that $d^*(v) < d^*(u) \leq key(u)$.

Proof. Let u be under-consistent but not under-estimated. Then, we have

$g_u(d(x_1), \ldots, d(x_k)) > d(u)$ (since u is under-consistent)
$\Rightarrow$ $g_u(d(x_1), \ldots, d(x_k)) > d^*(u)$ (since $d(u) \geq d^*(u)$ if u is not under-estimated)
$\Rightarrow$ u has an over-estimated predecessor v such that $d^*(v) < d^*(u)$ (from Lemma 3.2)

The result follows since $d^*(u) \leq d(u) = key(u)$ for an under-consistent vertex u that is not under-estimated. □

Lemma 3.4. If u is an over-consistent vertex, then u must be an over-estimated vertex or u must have an under-estimated predecessor v such that $d(v) < key(u) < d(u)$.

Proof. Let u be over-consistent but not over-estimated. Then, we have

$g_u(d(x_1), \ldots, d(x_k)) < d(u)$ (since u is over-consistent)
$\Rightarrow$ $g_u(d(x_1), \ldots, d(x_k)) < d^*(u)$ (since $d(u) \leq d^*(u)$ if u is not over-estimated)
$\Rightarrow$ $g_u(d(x_1), \ldots, d(x_k)) < g_u(d^*(x_1), \ldots, d^*(x_k))$
$\Rightarrow$ $d(x_i) < g_u(d(x_1), \ldots, d(x_k))$ and $d(x_i) < d^*(x_i)$ for some i (from Proposition 3.1)
$\Rightarrow$ $d(x_i) < key(u)$ and x_i is an underestimated predecessor of u.

The result follows since $key(u) = g_u(d(x_1), \ldots, d(x_k)) < d(u)$ for an over-consistent vertex u. □

Lemma 3.5. If u is an under-estimated vertex, then u must be an under-consistent vertex or u must have an under-estimated predecessor v such that $d(v) < d(u)$.

Proof. The proof is quite similar to the proof of Lemma 3.4. Let u be under-estimated but not under-consistent. Then, we have

$g_u(d(x_1), \ldots, d(x_k)) \leq d(u)$ (since u is not under-consistent)
$\Rightarrow$ $g_u(d(x_1), \ldots, d(x_k)) < d^*(u)$ (since $d(u) < d^*(u)$ if u is under-estimated)
$\Rightarrow$ $g_u(d(x_1), \ldots, d(x_k)) < g_u(d^*(x_1), \ldots, d^*(x_k))$
$\Rightarrow$ $d(x_i) < g_u(d(x_1), \ldots, d(x_k))$ and $d(x_i) < d^*(x_i)$ for some i (from Proposition 3.1)
$\Rightarrow$ $d(x_i) < d(u)$ and x_i is an underestimated predecessor of u.

□

We now establish some global conditions that must be satisfied for under-estimated and over-estimated vertices to exist, by repeated application of the previous lemmas concerning local conditions.

Lemma 3.6. If u is an under-estimated vertex then there exists an under-consistent vertex v such that $key(v) \leq d(u)$.

Proof.

Consider the sequence $u_0, u_1, \ldots, u_n$ of under-estimated vertices defined as follows. Let u_0 be the given under-estimated vertex u.

If u_i (which is guaranteed to be an under-estimated vertex by construction) is not an under-consistent vertex, then let u_{i+1} be an under-estimated predecessor of u_i such that $d(u_{i+1}) < d(u_i)$—such a vertex exists, as shown by Lemma 3.5.

If u_i is an under-consistent vertex, then we have $key(u_i) = d(u_i) < d(u_{i-1}) < \cdots < d(u)$, proving the lemma. Such an under-consistent vertex u_i will be reached eventually since the graph is finite and no vertex can be repeated in the sequence since the d values decrease monotonically. □

Lemma 3.7. If u is an over-estimated vertex then there exists an over-consistent vertex v such that $key(v) \leq d^*(u) < d(u)$.

Proof.

The proof is similar to that of the previous lemma. Consider the sequence $u_0, u_1, \ldots, u_n$ of over-estimated vertices defined as follows. Let u_0 be the given over-estimated vertex u.

If u_i is such that $g_{u_i}(d(x_1), \ldots, d(x_k)) > d^*(u_i)$ then let u_{i+1} be an over-estimated predecessor of u_i such that $d^*(u_{i+1}) < d^*(u_i)$—such a vertex exists, as shown by Lemma 3.2.

If u_i is such that $g_{u_i}(d(x_1), \ldots, d(x_k)) \leq d^*(u_i)$ then we have $d^*(u_i) < d(u_i)$ (since u_i is an over-estimated vertex by construction), and hence, $g_{u_i}(d(x_1), \ldots, d(x_k)) < d(u_i)$. In other words, we have an over-consistent vertex u_i such that $key(u_i) = g_{u_i}(d(x_1), \ldots, d(x_k)) \leq d^*(u_i) < \cdots < d^*(u_0)$, which proves the lemma. Again, the sequence must come to an end yielding the desired over-consistent vertex since the graph is finite. □

We are now ready to establish our main results that establish the correctness status of certain vertices from the consistency status of all the vertices.

Lemma 3.8. If no inconsistent vertex v exists such that $key(v) \leq d(u)$, then u is correct.

Proof. This follows immediately from Lemmas 3.6 and 3.7, since if u is incorrect it has to be either an under-estimated vertex or an over-estimated vertex. □

Lemma 3.9. If u is an over-consistent vertex and no under-consistent vertex w exists such that $key(w) < key(u)$, then u is an over-estimated vertex.

Proof. Let u be an over-consistent vertex. Lemma 3.4 says that if u is not an over-estimated vertex then u must have an under-estimated predecessor v such that $d(v) < key(u)$. But if v were an under-estimated vertex, then, from Lemma 3.6, there would exist an under-consistent vertex w such that $key(w) \leq d(v)$. The result follows. □

Lemma 3.10. If u is an under-consistent vertex and no over-consistent vertex w exists such that $key(w) < key(u)$, then u is an under-estimated vertex.

Proof. Let u be an under-consistent vertex. Lemma 3.3 says that if u is not an under-estimated vertex then u must have an over-estimated predecessor v such that $d^*(v) < key(u)$. But if v were an over-estimated vertex, then, from Lemma 3.7, there would exist an over-consistent vertex w such that $key(w) \leq d^*(v)$. The result follows. □

The following lemma shows that under certain conditions we can compute the correct d value of an incorrect vertex immediately.

Lemma 3.11. If u is an over-consistent vertex such that there exists no inconsistent vertex v with $key(v) < key(u)$ then $d^*(u) = key(u)$ (which is $g_u(d(x_1), \ldots, d(x_k))$ by definition).

Proof. We first show that $key(u) \leq d^*(u)$. It follows from Lemma 3.9 that u is an over-estimated vertex. However, Lemma 3.7 implies that there exists an over-consistent vertex x such that $key(x) \leq d^*(u)$. Since u has the least key among all inconsistent vertices $key(u) \leq key(x)$, and hence $key(u) \leq d^*(u)$.

We now show that $key(u) \geq d^*(u)$. For every vertex v define $d^+(v)$ as follows:

$$d^+(v) =_{def} \text{ if } (d(v) < key(u)) \text{ then } d(v) \text{ else } \infty.$$

Note that any vertex v for which $d(v) < key(u)$ must be correct from Lemma 3.8, since there can exist no inconsistent vertex w with $key(w) \leq d(v)$. Hence,

$$\begin{aligned} d^+(v) &= \text{if } (d(v) < key(u)) \text{ then } d(v) \text{ else } \infty \\ &= \text{if } (d(v) < key(u)) \text{ then } d^*(v) \text{ else } \infty \\ &\geq d^*(v) \end{aligned}$$

It follows from the definition of $d^+(v)$ that $key(u) = g_u(d(x_1), \ldots, d(x_k)) = g_u(d^+(x_1), \ldots, d^+(x_k))$ (using Proposition 3.1(a)) $\geq g_u(d^*(x_1), \ldots, d^*(x_k))$ (since g_u is monotonic) $\geq d^*(u)$. Hence, the result follows. □

The above lemmas suggest that we "process" the inconsistencies in order of increasing key value. If the inconsistent vertex with the least key value happens to be over-consistent then Lemma 3.11 shows how its correct value may be computed. If the inconsistent vertex with the least key value happens to be under-consistent then it follows from Lemma 3.10 that it is an under-estimated vertex. However, we have no analog of Lemma 3.11 that enables us to compute the correct value of such a vertex immediately. We process such a vertex by setting its value to be ∞, thereby converting it into either an over-consistent or a consistent vertex. The algorithm described in the following section is based on this idea.

4. The Algorithm

In this section we outline an algorithm for the dynamic SWSF fixed point problem. The algorithm is described as procedure *DynamicSWSF–FP* in Figure 1. We assume that a dependence graph G of a collection of SWSF equations is given, and that every vertex u in the graph has a tentative output value $d(u)$. We assume that the set U of vertices whose associated equations have been modified is also part of the input to the algorithm. In other words, only vertices in U may be inconsistent. The other vertices are guaranteed to be consistent. This is the precondition for the algorithm to compute the correct solution to the modified set of equations.

The idea behind the algorithm was explained in the previous section. The algorithm maintains the following invariants, and the steps in the algorithm can be understood easier in terms of the invariants. The algorithm maintains a heap of all the inconsistent vertices—both over-consistent and under-consistent vertices—in the graph. An overconsistent vertex u occurs in the heap with a key (priority) of $g_u(d(x_1), \ldots, d(x_k))$, while an under-consistent vertex u occurs in the heap with a key value of $d(u)$. The heap is used to identify the inconsistency with the least key value at every stage. For every inconsistent vertex u, the algorithm also maintains $rhs(u)$, the value of the right-hand side of the equation associated with vertex u. Let us say a vertex u ***satisfies the invariant*** if (a) u occurs in *Heap* with key k iff u is an inconsistent vertex with $key(u) = k$, and (b) if u is an inconsistent vertex then $rhs(u) = g_u(d(x_1), \ldots, d(x_k))$.

The algorithm makes uses of the heap operations $InsertIntoHeap(Heap, item, key)$ and $ExtractAndDeleteMin(Heap)$, which need no explanation. It also uses a heap operation $AdjustHeap(Heap, i, k)$ that inserts an item i into *Heap* with key k if i is not in *Heap* already, and changes the key of item i in *Heap* to k if i is in *Heap* already.

We now verify that the algorithm does indeed maintain the invariants described above. Thus, we first need to show that all vertices satisfy the invariant whenever execution reaches line [8]. The precondition guarantees that all the initially inconsistent vertices must be in U. In lines [1]-[7], the algorithm creates a heap out of all the initially inconsistent vertices in the graph, and simultaneously the value $rhs(u)$ is properly defined for every inconsistent vertex u. Hence the invariant holds when execution reaches line [8] for the first time.

The loop in lines [8]-[31] processes and "fixes" the inconsistencies in the graph one by one, in increasing order of key value. An over-consistent vertex u is processed (lines [11]-[19]) by updating $d(u)$ to equal $g_u(d(x_1), \ldots, d(x_k))$, the value of the right-hand side of the equation associated with vertex u. In view of Lemma 3.11, this converts the over-estimated vertex u into a correct vertex. As a result of the assignment of a new value to $d(u)$ in line [11] some of the successors of u may fail to satisfy the invariant, though any vertex which is not a successor of u will continue to satisfy the invariant. When the loop in lines [12]-[19] completes execution all vertices are guaranteed to satisfy the invariant. In particular, lines [13]-[18] make sure v satisfies the invariant by computing its *rhs* value, determining its consistency status, and adjusting the heap.

```
procedure DynamicSWSF-FP (G, U)
declare
        G : a dependence graph of a set of SWSF equations
        U : the set of modified vertices in G
        u, v, w: vertices
        Heap: a heap of vertices
preconditions
        Every vertex in V(G)–U is consistent
begin
[1]     Heap := ∅
[2]     for u ∈ U do
[3]         rhs(u) := g_u(d(x_1), . . . , d(x_k))
[4]         if rhs(u) ≠ d(u) then
[5]             InsertIntoHeap( Heap, u, min(rhs(u), d(u)))
[6]         fi
[7]     od
[8]     while Heap ≠ ∅ do
[9]         u := ExtractAndDeleteMin( Heap )
[10]        if rhs(u) < d(u) then /* u is overconsistent */
[11]            d(u) := rhs(u)
[12]            for v ∈ Succ(u) do
[13]                rhs(v) := g_v(d(x_1), . . . , d(x_k))
[14]                if rhs(v) ≠ d(v) then
[15]                    AdjustHeap(Heap, v, min(rhs(v), d(v)))
[16]                else
[17]                    if v ∈ Heap then Remove v from Heap fi
[18]                fi
[19]            od
[20]        else /* u is underconsistent */
[21]            d(u) := ∞
[22]            for v ∈ (Succ(u) ∪ {u}) do
[23]                rhs(v) := g_v(d(x_1), . . . , d(x_k))
[24]                if rhs(v) ≠ d(v) then
[25]                    AdjustHeap(Heap, v, min(rhs(v), d(v)))
[26]                else
[27]                    if v ∈ Heap then Remove v from Heap fi
[28]                fi
[29]            od
[30]        fi
[31]    od
end
postconditions
        Every vertex in V(G) is consistent
```

Figure 1. An algorithm for the dynamic SWSF fixed point problem.

An under-consistent vertex u is processed (lines [21]-[30]) by updating $d(u)$ to equal ∞, followed by an appropriate updating of the heap. In view of Lemma 3.10, this step converts an under-estimated vertex into either an over-estimated vertex or a correct vertex. Following the assignment of a new value to $d(u)$ in line [21], only u or some successor of u can fail to satisfy the invariant. These vertices are appropriately processed in lines [22]-[29], and hence the invariant is satisfied whenever execution reaches line [8].

To understand how the algorithm makes progress towards the correct solution consider how the correctness status of the vertices in the graph change. In each iteration of the loop in lines [8]-[31] the value, and hence the correctness status, of only one vertex (namely u) changes. In particular, in each iteration exactly

one of the following happens. (1) An over-estimated vertex becomes correct. (2) An under-estimated vertex becomes over-estimated. (3) An under-estimated vertex becomes correct. In particular, the value of a correct vertex is never changed. An initially (*i.e.*, at the beginning of the algorithm) over-estimated vertex changes value exactly once. An initially under-estimated vertex changes values at most twice (either to the correct value ∞, or first to ∞ and then to the correct final value). It follows that the algorithm must terminate.

Since the heap is empty when the algorithm terminates, it follows immediately from the loop invariant that there exists no inconsistency in the graph when the algorithm halts. In particular, the computed d values form the unique fixed point of the collection Q of equations.

Let us now determine the time complexity of the algorithm. Recall that C_δ is a bound on the time required to compute the function associated with any vertex in $Changed \cup Succ(Changed)$. The initialization in lines [1]-[7] involves $|U|$ function evaluations and $|U|$ heap operations (insertions) and consequently takes $O(|U| \cdot (C_\delta + \log|U|))$ time, which is $O(|\delta| \cdot (C_\delta + \log|\delta|))$ time since U is $Modified_\delta$.

Every vertex that is in the heap at some point during the execution must be an affected vertex or the successor of an affected vertex. Hence, the maximum number of elements in the heap at any point is $O(\|\delta\|)$, and every heap operation takes $O(\log\|\delta\|)$ time. It follows from the explanation given earlier that lines [11]-[19] are executed at most once for each affected vertex u. In these lines, the function associated with every vertex in $Succ(u)$ is evaluated once, and at most $|Succ(u)|$ heap operations are performed. Hence, the lines [11]-[19] take $O(\|\{u\}\| \cdot (C_\delta + \log\|\delta\|))$ time (in one iteration). Lines [20]-[30] are similarly executed at most once for each affected vertex u. Consequently, lines [20]-[30] also take time $O(\|\{u\}\| \cdot (C_\delta + \log\|\delta\|))$ time (in one iteration).

Consequently, the whole algorithm runs in time $O(\|\delta\| \cdot (\log\|\delta\| + C_\delta))$, and the algorithm is a bounded scheduling cost algorithm.

5. An Improved Algorithm

The algorithm presented in the previous section is not the most efficient incremental algorithm for the SSSP>0 problem. The source of inefficiency is that the algorithm assumes that each function g_i is an *s.w.s.f.* and no more. The functions that arise in the shortest-path problem (and in any SSF grammar problem), however, have a special form. The function corresponding to a vertex u other than the source is $\min_{v \in Pred(u)} [d(v) + length(u \to v)]$. Such expressions permit the possibility of incremental computation of the expression itself. For instance, evaluating this value from scratch takes time $\Theta(|Pred(u)|)$, while if the value of this expression is known, and the value of $d(v)$ decreases for some $v \in Pred(u)$, the new value of the expression can be recomputed incrementally in constant time. Note that this kind of incremental recomputation of an expression's value is performed repeatedly in Dijkstra's algorithm for the batch SSSP≥0 problem. Unfortunately, an incremental algorithm for the SSSP problem has to also contend with the possibility that the value of $d(v)$ *increases* for some $v \in Pred(u)$. The need to maintain the value of the expression $\min_{v \in Pred(u)} [d(v) + length(u \to v)]$ as the values of $d(v)$ change immediately suggests the possibility of maintaining the set of all values $\{ d(v) + length(u \to v) \mid v \in Pred(u) \}$ as a heap. Our approach is to maintain a particular subset of the set $\{ d(v) + length(u \to v) \mid v \in Pred(u) \}$ as a heap, since maintaining the whole set as a heap requires unnecesary work.

In this section we present a more efficient version of algorithm *DynamicSWSF–FP* that utilizes the special form of the equations induced by the SSF grammar problem. The algorithm is described as procedure *DynamicSSF–G* in Figure 2. The algorithm, as presented, addresses the dynamic SSF grammar problem, and, hence, might appear to be less general than the algorithm presented in the previous section, which

```
procedure DynamicSSF-G (G, P)
declare
        G : a SSF grammar
        P : the set of modified productions in G
        GlobalHeap: a heap of non-terminals
preconditions
        Every production in G–P is consistent. (See Definition 5.1)

        procedure recomputeProductionValue(p)
        declare
          p : a production
        begin
[1]       let p be the production Y ⟶ g(X_1, . . . , X_k)
[2]       value = g(d(X_1), . . . , d(X_k))
[3]       if (value < d(Y)) then
[4]         AdjustHeap( Heap(Y), p, value)
[5]       else
[6]         if p ∈ Heap(Y) then Remove p from Heap(Y) fi
[7]       fi
[8]       if (value ≤ d(Y)) then SP(Y) := SP(Y) ∪ {p} else SP(Y) := SP(Y)–{p} fi
[9]       if (SP(Y) = ∅) then     /* Y is under-consistent */
[10]        AdjustHeap( GlobalHeap, Y, d(Y))
[11]      elseif Heap(Y) ≠ ∅ then     /* Y is over-consistent */
[12]        AdjustHeap( GlobalHeap, Y, min–key(Heap(Y)))
[13]      else    /* Y is consistent */
[14]        if Y ∈ GlobalHeap then Remove Y from GlobalHeap fi
[15]      fi
        end

begin
[16]    GlobalHeap := ∅
[17]    for every production p ∈ P do
[18]        recomputeProductionValue(p)
[19]    od
[20]    while GlobalHeap ≠ ∅ do
[21]        Select and remove from GlobalHeap a non-terminal X with minimum key value
[22]        if key(X) < d(X) then /* X is overconsistent */
[23]            d(X) := key(X)
[24]            SP(X) := { p | p is a production for X such that value(p) = d(X) }
[25]            Heap(X) := ∅
[26]            for every production p with X on the right-hand side do
[27]                recomputeProductionValue(p)
[28]            od
[29]        else /* X is underconsistent */
[30]            d(X) := ∞
[31]            SP(X) := { p | p is a production for X }
[32]            Heap(X) := makeHeap({ p | p is a production for X with value(p) < d(X) })
[33]            if Heap(X) ≠ ∅ then AdjustHeap( GlobalHeap, X, min–key(Heap(X))) fi
[34]            for every production p with X on the right-hand side do
[35]                recomputeProductionValue(p)
[36]            od
[37]        fi
[38]    od
end
postconditions
        Every non-terminal and production in G is consistent
```

Figure 2. An algorithm for the dynamic SSF grammar problem.

addresses the dynamic SWSF fixed point problem. Procedure *DynamicSSF–G can*, in fact, be used for the dynamic SWSF fixed point problem with some simple modifications, though it will be more efficient than procedure *DynamicSWSF–FP* only when the equations have the special form described above. We address the less general SSF grammar problem here since it is this problem that motivates the improvements to the algorithm, but emphasize that the improved algorithm is as general as the original algorithm in terms of the class of problem instances that it can handle. (It is because of the latter reason that we refer to procedure *DynamicSSF–G* as an improvement of *DynamicSWSF–FP* rather than merely a specialization of *DynamicSWSF–FP*.)

We first explain the idea behind the algorithm, then prove the correctness of the algorithm, and finally analyze its time complexity.

We assume that an SSF grammar is given, and that every non-terminal X in the grammar has a tentative output value $d(X)$. We assume that the change in the input takes the form of a change in some of the productions and production functions of the grammar. This type of modification is general enough to include insertions and deletions of productions as well, since a non-existent production can be treated as a production whose production function is the constant-valued function ∞. The insertion or deletion of non-terminals can be handled just as easily. So we assume that the input to the algorithm includes a set P of productions whose production functions have been modified.

The steps given in lines [16]-[38] implement essentially the same idea as procedure *DynamicSWSF–FP*. A heap, called *GlobalHeap*, of all the inconsistent non-terminals is maintained as before, and in each iteration the inconsistent non-terminal X with the least key is processed, just as before. In *DynamicSWSF–FP* a change in the value of a vertex was followed by the complete re-evaluation of the function associated with the successors of that vertex, in order to identify the change in the consistency status of those vertices. This is the step that the new algorithm, procedure *DynamicSSF–G*, performs differently. The new algorithm identifies changes in the consistency status of other non-terminals in an *incremental* fashion. We now describe the auxiliary data structures that the algorithm uses to do this. These auxiliary data structures are retained across different invocations of the procedure.

Note that the value associated with a non-terminal X is $d(X)$. We define the *value* of a production $Y \longrightarrow g(X_1, \ldots, X_k)$ to be $g(d(X_1), \ldots, d(X_k))$. For every non-terminal X, the algorithm maintains a set $SP(X)$ of all productions with X as the left-hand side whose value is less than or equal to $d(X)$. Actually we need to maintain only the cardinality of this set, but we use the set itself since it makes the algorithm easier to understand. The algorithm also maintains for every non-terminal X a heap $Heap(X)$ of all the productions with X as the left-hand side whose value is strictly less than $d(X)$, with the value of the production being its key in the heap.

Consider a production $p = Y \longrightarrow g(X_1, \ldots, X_k)$. We say that the production p ***satisfies the invariant*** if (a) $p \in SP(Y)$ iff $value(p) \leq d(Y)$ and (b) $p \in Heap(Y)$ iff $value(p) < d(Y)$. Thus, we want to maintain $SP(Y)$ and $Heap(Y)$ such that all productions satisfy the invariant. However, both at the beginning of the update and temporarily during the update, several productions may fail to satisfy the invariant.

We use these auxiliary data structures to determine the consistency status of non-terminals. Note that a non-terminal X is under-consistent iff $SP(X)$ is empty and $d(X) < \infty$,[5] in which case its key is $d(X)$; X is over-consistent iff $Heap(X)$ is non-empty, in which case its key is given by $min\text{–}key(Heap(X))$, the key of

[5]In general, the condition that $SP(X)$ be empty subsumes the condition that $d(X)$ be less than ∞. The latter condition is relevant only if no production has X on the left-hand side.

the item with the minimum key value in $Heap(X)$. The invariant that *GlobalHeap* satisfies is that every non-terminal X for which $SP(X)$ is empty and $d(X)$ is less than ∞ occurs in *GlobalHeap* with a key of $d(X)$, while every non-terminal X for which $Heap(X)$ is non-empty occurs in *GlobalHeap* with a key of $min\text{–}key(Heap(X))$. It follows from the preceeding explanation that *GlobalHeap* consists of exactly the inconsistent non-terminals with their appropriate keys.

We now show that the algorithm maintains these data structures correctly and that it updates the solution correctly. However, we first need to understand the precondition these data structures will have to satisfy at the beginning of the algorithm.

Definition 5.1. A production $p = Y \longrightarrow g(X_1, \ldots, X_k)$ is said to be ***consistent*** if (a) $p \notin Heap(Y)$ and (b) either $value(p) = d(Y)$ and $p \in SP(Y)$ or $value(p) > d(Y)$ and $p \notin SP(Y)$. In other words, p is consistent iff it satisfies the invariant and, in addition, $value(p) \geq d(Y)$.

The precondition we assume to hold at the beginning of the update is that every unmodified production is consistent. The invariant the algorithm maintains is that whenever execution reaches line [20] every production satisfies the invariant, and that the *GlobalHeap* contains exactly the inconsistent non-terminals. The postcondition established by the algorithm is that every production and non-terminal in the grammar will be consistent.

The procedure *recomputeProductionValue(p)* makes production p consistent by evaluating its value (in line [2]) and updating the data structures $SP(Y)$ (line [8]) and $Heap(Y)$ (lines [3]-[7]) appropriately, where Y is the left-hand side of p. These changes are followed by appropriate updates to *GlobalHeap* in lines [9]-[15]. Note that some of these steps can be redundant, in the sense that they do nothing. For instance, if the value of the production does not change, then steps [2]-[15] will not make any change to the data structures.

We now show that whenever execution reaches line [20] every production satisfies the invariant, and *GlobalHeap* contains exactly the inconsistent non-terminals. The lines [16]-[19] initially establish the invariant. Subsequently, in each iteration of the loop in lines [20]-[38], whenever the value of a non-terminal changes (either in line [23] or line [30]) procedure *recomputeProductionValue(p)* is called for every production p that might have become inconsistent. Thus, the invariant is re-established.

It follows from the explanation in the previous paragraph that every non-terminal and production in the grammar is consistent when the algorithm halts.

Let us now consider the time complexity of the improved algorithm. In Section 4, algorithm *DynamicSWSF–FP* dealt with the dependence graph of a collection of SWSF equations. There, the individual equations were treated as indivisible units in that an equation was the smallest unit of the input that could be modified. The algorithm outlined in this section, however, specifically deals with the equations generated by an SSF grammar. A finer granularity of input modifications is made possible by allowing individual productions to be modified. Consequently, it is necessary to consider a refined version of the dependence graph in analyzing the time complexity of the algorithm.

The bipartite graph $B = (N,P,E)$ consists of two disjoint sets of vertices N and P, and a set of edges E between N and P. The set N consists of a vertex n_X for every non-terminal X in the grammar, while the set P consists of a vertex n_p for every production p in the grammar. For every production p in the grammar, the graph contains an edge $n_X \rightarrow n_p$ for every non-terminal X that occurs on the right-hand side of p, and an edge $n_p \rightarrow n_Y$ where Y is the left-hand side non-terminal of p. The set *Affected* consists of the set of all vertices n_X where X is a non-terminal whose output value changes, while the set *Modified* consists of the set of all vertices n_p, where p is a modified production. The set *Changed* is $Affected \cup Modified$.

Let us first consider the time spent in the main procedure, namely lines [16]-[38]. As explained in the previous section, the loop in lines [20]-[38] iterates at most $2 \cdot |Affected|$ times. Lines [23]-[28] are executed at most once for every affected non-terminal X, while lines [30]-[36] are similarly executed at most once for every affected non-terminal X. Consequently, the steps executed by the main procedure can be divided into (a) $O(\|Changed\|_B)$ invocations of the procedure *recomputeProductionValue* (lines [18], [27] and [35]), (b) $O(|Affected|)$ operations on *GlobalHeap* (line [21]), and (c) the remaining steps, which take time $O(\|Changed\|_B)$.

Let us now consider the time taken by a single execution of procedure *recomputeProductionValue*. The procedure essentially performs (a) one function computation (line [2]), (b) $O(1)$ set operations (lines [8] and [9]), (c) $O(1)$ *Heap*(Y) operations (lines [4] or [6]), and (d) $O(1)$ *GlobalHeap*(Y) operations (lines [10], [12] or [14]). The set operations on $SP(Y)$ can be done in constant time by associating every production $Y \longrightarrow g(X_1, \ldots, X_k)$ with a bit that indicates if it is in the set $SP(Y)$ or not. It can be easily verified that each *Heap*(Y) and *GlobalHeap* have at most $\|Affected\|_B$ elements. Consequently, each heap operation takes at most $\log \|Affected\|_B$ time.

As before, $C_{B,\delta}$ is a bound on the time required to compute the production function associated with any production in $Changed \cup Succ(Changed)$. Then, procedure *recomputeProductionValue* itself takes time $O(\log \|\delta\|_B + C_{B,\delta})$. Hence, the whole algorithm runs in time $O(\|\delta\|_B \cdot (\log \|\delta\|_B + C_{B,\delta}))$.

Let us now consider the SSSP>0 problem. Each production function can be evaluated in constant time in this case, and, hence, the algorithm runs in time $O(\|\delta\| \log \|\delta\|)$. (Note that in the case of the SSSP>0 problem the input graph G and the bipartite graph B are closely related, since each "production" vertex in B corresponds to an edge in G. Hence, $\|\delta\|_B = O(\|\delta\|_G)$.)

We now consider a special type of input modification for the SSSP>0 problem for which it is possible to give a better bound on the time taken by the update algorithm. Assume that the change in the input is a homogeneous decrease in the length of one or more edges. In other words, no edges are deleted and no edge-length is increased. In this case it can be seen that no under-consistent vertex exists, and that the value of no vertex increases during the update. In particular, the *AdjustHeap* operations (in lines [4], [10], and [12]) either perform an insertion or decrease the key of an item. Lines [6] and [14] are never executed. Consequently, procedure *recomputeProductionValue* takes time $O(1)$ if relaxed heaps [8] or Fibonacci heaps [11] are used. (In the latter case, the time complexity is the amortized complexity.) It can also be verified that the number of elements in any of the heaps is $O(|\delta|)$. Hence, the algorithm runs in time $O(\|\delta\| + |\delta| \log |\delta|)$. In particular, if m edges are inserted into an empty graph with n vertices, the algorithm works exactly like the $O(m + n \log n)$ implementation of Dijkstra's algorithm due to Fredman and Tarjan [11]. The asymptotic complexity of the algorithm can be further improved by using the recently developed AF-heap data structure [12].

6. Extensions to the Algorithm

In this section we briefly outline various possible extensions and applications of the incremental algorithm described in the previous section.

6.1. Maintaining Minimum Cost Derivations

We have so far considered only the problem of maintaining the *cost* of the minimum cost derivations, and not the problem of maintaining minimum cost *derivations* themselves. However, the algorithm outlined in the previous section can be easily extended to maintain the minimum cost derivations too. The set $SP(X)$ computed by the algorithm is the set of all productions for X that can be utilized as the first production in

minimum cost derivations of terminal strings from X. Hence, all possible minimum cost derivations from a non-terminal can be recovered from this information. In particular, consider the SSSP>0 problem. Every production p for a non-terminal N_v corresponds to an incoming edge $u \rightarrow v$ of vertex v, where v is a vertex other than the source. The production p will be in $SP(N_v)$ iff a shortest path from the source to u followed by the edge $u \rightarrow v$ yields a shortest path from the source to v. Hence, a single shortest-path from the source vertex to any given vertex can be identified in time proportional to the number of edges in that path, provided the set $SP(X)$ is implemented so that an arbitrary element from the set can be chosen in constant time. As explained earlier, the various sets $SP(X)$ can be implemented by associating a bit with every edge. If the set of all edges in a set $SP(X)$ are also combined into a doubly linked list, then an arbitrary element from the set can be chosen in constant time.

6.2. The All-Pairs Shortest-Path Problem

We have seen that the algorithm outlined in the previous section can be utilized in updating the solution to the single-source (or the single-sink) shortest-path problem when the underlying graph undergoes modifications. We briefly sketch how this algorithm can be adapted to update the solution to the all-pairs shortest-path problem too. The essential approach is to make repeated use of our incremental algorithm for the SSSP>0 problem. However, it is not necessary to update the single-source solution for every vertex in the graph; it is possible to identify a subset of the vertices for which it is sufficient to update the single-source solution. Let $u_i \rightarrow v_i$, for $1 \leq i \leq k$, be the set of modified (inserted or deleted) edges. Let $d(x,y)$ denote the length of a shortest path from x to y. Then, for any two vertices s and t, $d(s,t)$ can change only if for some $i \in [1,k]$ both $d(s,v_i)$ and $d(u_i,t)$ change. Hence, by updating the single-source solution for every u_i, we can identify the set of vertices t for which the single-sink solution will change. Similarly, by updating the single-sink solution for every v_i, we can identify the set of vertices s for which the single-source solution will change. Then, we can update the single-sink solution and the single-source solution only for those vertices for which the solution can change. However, we note that for certain special cases, such as updating the solution to the APSP>0 problem after the insertion of an edge, this approach does not yield the best possible incremental algorithm. (See [23], for instance.)

6.3. Handling Edges with Non-Positive Lengths

The proof of correctness of our algorithm and the analysis of its time complexity both rely on the fact that all edges have a positive length. We now discuss some types of input changes for which this restriction on the edge lengths can be somewhat relaxed. We first consider zero-length edges. It can be shown that if the change in the input graph is a homogeneous decrease in the length of one or more edges then the algorithm works correctly as long as all edges have a non-negative length (*i.e.*, zero-length edges do not pose a problem). Similarly, if the input change is a homogeneous increase in the length of one or more edges then the algorithm works correctly as long as all edges have a non-negative length and there are no cycles in the graph of zero length (*i.e.*, zero-length edges do not pose a problem as long as no zero-length cycles exist in the graph).

We now consider negative length edges. For certain types of input modifications it is possible to use a variant of our incremental algorithm to update the solution to the SSSP problem (with arbitrary edge lengths), as long as all cycles in the graph have a positive length. The idea is to adapt the technique of Edmonds and Karp for transforming the length of every edge to a non-negative real without changing the graph's shortest paths [9, 27]. Their technique is based on the observation that if f is any function that maps vertices of the graph to reals, and the length of each edge $a \rightarrow b$ is replaced by $f(a) + length(a \rightarrow b) - f(b)$, then the shortest paths in the graph are unchanged from the original edge-length mapping. If f satisfies the property that $f(a) + length(a \rightarrow b) - f(b) \geq 0$ for every edge $a \rightarrow b$ in

the graph, then the transformed length of every edge will be positive.

Now consider the incremental SSSP problem. Let $d_{old}(u)$ denote the length of the shortest path in the input graph G from $source(G)$ to u before G was modified. Consider the effect of the above edge-length transformation if we simply define $f(u)$ to be $d_{old}(u)$. First note that the transformation is well-defined only for edges $a \rightarrow b$ such that $d_{old}(b)$ is not ∞. For every edge $a \rightarrow b$ in the original graph we have $d_{old}(b) \leq d_{old}(a) + length_{old}(a \rightarrow b)$. Consequently, $d_{old}(a) + length_{old}(a \rightarrow b) - d_{old}(b) \geq 0$. Hence, the transformed length of an edge $a \rightarrow b$ will be non-negative as long as $length_{new}(a \rightarrow b) \geq length_{old}(a \rightarrow b)$ (*i.e.*, as long as the length of the edge $a \rightarrow b$ was not decreased during the input modification), and $d_{old}(b)$ is not ∞.

In particular, this idea can be used to adapt our incremental algorithm to update the solution to the SSSP problem when the lengths of a collection of edges are increased (possibly to ∞), and no edge is inserted or no edge-length is decreased. This will work since the length of an edge $a \rightarrow b$ is relevant only if a can be reached from the source vertex and, hence, only if both $d_{old}(a)$ and $d_{old}(b)$ are finite. The transformed length of all such edges are non-negative, and our incremental algorithm is applicable as long as there are no cycles of zero length in the graph. Note that it is not necessary to compute the transformed length for all edges at the beginning; instead, the transformed length of an edge can be computed as and when the length of that edge is needed. This is essential to keep the algorithm a bounded one.

The technique of edge-length transformation can also be used in a special case of edge insertion or edge-length decrease. Assume that the length of a set of edges F, all directed to a specific vertex u that was already reachable from the source, are decreased (possibly from ∞). The above edge-length transformation makes the lengths of all *relevant* edges non-negative. The transformed length of the edges in F are not guaranteed to be non-negative; however, this causes no difficulties because edges directed to u are in a sense irrelevant to the updating algorithm. We leave the details to the reader.

6.4. The Batch Shortest-Path Problem in the Presence of Few Negative Edges

Yap [28] describes an algorithm for finding the shortest path between two vertices in a graph that may include edges with negative length. This algorithm works better than the standard Bellman and Ford algorithm when the number of negative length edges is small. An algorithm with a slightly better time complexity can be obtained by making use of the incremental algorithms for the SSSP problem. The latter algorithm in fact solves the single-source or single-sink problem rather than just the single-pair problem.

We first consider the time complexity of Yap's algorithm. Let G be the given graph. Let n denote the number of vertices in G and let m denote the number of edges in G. Let h denote the number of edges whose length is negative, and let k denote $min(h,n)$. Yap's approach reduces a single-pair shortest path problem on the given graph G to $\min(h+1,n)$ SSSP$\geq$0 problems on the subgraph of G consisting of only non-negative edges, and a single pair shortest path problem on a graph consisting of $O(k)$ vertices and $O(k^2)$ edges of arbitrary (that is, both positive and negative) lengths. This yields an $O(k[m+n\log n]+k^3)$ algorithm for the problem, which is better than the standard $O(mn)$ algorithm for sufficiently small k. (Actually, Yap describes the time complexity of the algorithm as $O(kn^2)$, since he makes use of Dijkstra's $O(n^2)$ algorithm. The above complexity follows from Fredman and Tarjan's [11] improvement to Dijkstra's algorithm. The complexity of the above algorithm can be improved slightly by utilising the recent $O(m+n\log n/\log\log n)$ shortest path algorithm due to Fredman and Willard [12]).

We now consider how our incremental algorithm for the shortest-path problem can be used to solve this problem better. Let $u_1, \ldots, u_{k'}$ be the set of all vertices in the graph that have an incoming edge of negative length. Thus $k' \leq k$. First replace all the negative edges in the given graph G with zero weight edges.

Compute the solution to this graph by using, say, the Fredman-Tarjan improvement to Dijkstra's algorithm. Now process the vertices $u_1, \ldots, u_{k'}$ one by one. The vertex u_i is processed by restoring the length of all the edges directed to u_i to their actual value and updating the solution using the adaptation of our incremental algorithm explained in Section 6.3.

The updating after each insertion step takes $O(m + n \log n)$ time in the worst case. Hence, the algorithm runs in time $O(k'[m + n \log n])$. In general, the algorithm can be expected to take less time than this bound, since all the update steps have bounded complexity.

7. Related Work

In this paper we have presented an incremental algorithm for the dynamic SWSF fixed point problem. The dynamic SWSF fixed point problem includes the dynamic SSF grammar problem as a special case, which, in turn, includes the dynamic SSSP>0 problem as a special case. Thus, we obtain an incremental algorithm for the dynamic SSSP>0 problem as a special case of algorithm *DynamicSSF−G*, which was described in Section 5. We have also described how the algorithm can be generalized to handle negative edge lengths under certain conditions, and how the algorithm for the dynamic single-source shortest-path problem can be utilized for the dynamic all-pairs shortest-path problem as well.

Knuth [17] introduced the grammar problem as a generalization of the shortest-path problem, and generalized Dijkstra's algorithm to solve the batch SF grammar problem. We know of no previous work on incremental algorithms for the dynamic grammar problem.

Previous work on algorithms for the dynamic shortest-path problem include papers by Murchland [20, 21], Loubal [19], Rodionov [24], Halder [15], Pape [22], Hsieh *et al.* [16], Cheston [4], Dionne [7], Goto *et al.* [14], Cheston and Corneil [5], Rohnert [25], Even and Gazit [10], Lin and Chang [18], Ausiello *et al.* [1, 2], and Ramalingam and Reps [23]. These algorithms may be classified into groups based on (a) the information computed by the algorithm (such as the whether the all-pairs or single-source version of the problem is addressed), (b) the assumptions made about the edge lengths, and (c) the type of modification that the algorithm handles. What distinguishes the work reported in this paper from all of the work cited above is that it is the first incremental algorithm that places no restrictions on how the underlying graph can be modified between updates. This work addresses the single-source shortest-path problem, with the restriction that all edges be positive. Section 6 discusses several extensions and refinements of this work.

The remainder of this section provides a brief overview of the different groups of dynamic shortest-path algorithms, the different techniques utilized by the various algorithms, and a brief comparison of the different algorithms. The table in Figure 3 summarizes this discussion. We remind the reader that our comments about the cases of an edge-insertion or an edge-deletion apply equally well to the cases of a decrease in an edge length and an increase in an edge length, respectively.

We begin with the version of the problem that has been studied the most, namely the all-pairs version. Given a graph G and a modification δ to the graph, let $d_{old}(x,y)$ and $d_{new}(x,y)$ denote the length of a shortest path from x to y in the graphs G and $G + \delta$ respectively. The pair (x,y) is said to be an *affected pair* if $d_{new}(x,y)$ is different from $d_{old}(x,y)$. A vertex x is said to be an *affected source* if there exists a vertex y such that (x,y) is an affected pair; similarly x is said to be an *affected sink* if there exists a vertex y such that (y,x) is an affected pair.

Let us now consider the problem of processing the insertion of an edge $u \rightarrow v$. This problem is in some sense the easiest among the various versions of the dynamic shortest-path problem; at least, it is fairly straight-forward to determine $d_{new}(x,y)$ in constant time, for any given pair of vertices (x,y) since

Problem	Modifications	Best bounded algorithm(s)	Other bounded algorithms	Unbounded algorithms
APSP	Single Edge Insertion	[18], [2]	[23], [25], [10]	[7], [24], [19], [20]
APSP	Single Edge Deletion			[25], [10], [7], [24], [20]
APSP-Cycle>0	Single Edge Deletion	[23]		
APSP>0	Single Edge Deletion			[15]
APSP>0	Arbitrary Modification	This paper		Repeated applications of algorithms for unit changes
Multiple SSSP	Multiple Edges Insertion			[14]
SSSP>0	Arbitrary Modification	This paper		
SSSP-Cycle>0	Multiple Edges Deletion	This paper		
SSSP	Restricted Edge Insertion	This paper		[14]

Figure 3. Various versions of the dynamic shortest-path problem and incremental algorithms for them. Note that APSP>0 and SSSP>0 refer to problems where every edge is assumed to have positive length, while APSP-Cycle>0 and SSSP-Cycle>0 refer to problems where every cycle is assumed to have positive length, with no restrictions on edge lengths. The modification referred to in the last item of the table, namely "restricted edge insertion", is the insertion of one or more edges, all directed to the same vertex, a vertex that must already be reachable from the source.

$$d_{new}(x,y) = \min\,(\, d_{old}(x,y),\, d_{old}(x,u) + length_{new}(u \rightarrow v) + d_{old}(v,y)\,). \qquad (\ddagger)$$

Computing $d_{new}(x,y)$ for every pair of vertices (x,y) using the above equation takes time $O(n^2)$, which is better than the time complexity of the best batch algorithm for APSP. Most of the known algorithms for this problem do even better by first identifying an approximation A to the set of all affected pairs and then updating $d(x,y)$ only for $(x,y) \in A$. The best algorithm currently known for this problem, developed independently by Lin and Chang [18] and Ausiello et al. [2], restricts the set of pairs of vertices for which the d value is recomputed by a careful traversal of the shortest-path trees of the graph before the modification. The algorithm due to Even and Gazit [10] is similar and identifies the same set of pairs of vertices but is slightly less efficient since it does not maintain shortest-path trees. The algorithms due to Rohnert [25] and Ramalingam and Reps [23] are based on similar ideas, though they are not as efficient as the algorithms of [18] and [2]. All of the above algorithms are bounded algorithms. It is worth mentioning that the improved efficiency of the algorithms described in [18] and [2] is obtained at a cost: these algorithms make use of the shortest-path-tree data structure, the maintenance of which can make the processing of an *edge-deletion* more expensive. The algorithms due to Murchland [20], Dionne [7], and Cheston [4] are all based on the observation that x is an affected source [sink] iff (x,v) $[(u,x)]$ is an affected pair. These algorithms identify the set of affected sources S_1 and the set of affected sinks S_2 in $O(n)$ time using equation (‡), and use $S_1 \times S_2$ as an approximation to the set of affected pairs. Consequently, these algorithms are unbounded.

Let us now consider the problem of processing the deletion of an edge $u \rightarrow v$ from the graph. Edge deletion is not as easy to handle as edge insertion. As Spira and Pan [26] show, the batch all-pairs shortest-path problem can, in some sense, be reduced to the problem of updating the solution to the all-pairs shortest-path problem after an edge deletion. An incremental algorithm that saves only the shortest-paths information cannot, in the worst-case, do any better than a batch algorithm, which is not true in the case of edge

insertion.

Most algorithms for processing an edge deletion follow the approach of first identifying an approximation A to the set of all affected pairs, and then computing the new d value for every affected pair. Ramalingam and Reps [23] show that it is possible to identify the set of affected pairs exactly if the graph does not have zero-length cycles, and describe the only known bounded incremental algorithm for this problem. This algorithm is based on the repeated application of a bounded algorithm for the dynamic SSSP>0 problem (see below). The set of all affected sinks is identified by using the algorithm for the dynamic SSSP>0 problem with u as the source, since x is an affected sink iff (u,x) is an affected pair. The APSP solution can then be updated by updating the single-sink solution for every affected sink.

The algorithms due to Rohnert [25] and Even and Gazit [10] can also be viewed as consisting of the repeated application of an algorithm for the dynamic SSSP problem, though they are not described as such. These algorithms, however, do not identify the set of affected pairs exactly. A vertex pair (x,y) is treated as a possibly affected pair iff $u \rightarrow v$ is in the current shortest path from x to y that the algorithm maintains. (Note that an alternative shortest path from x to y that does not contain edge $u \rightarrow v$ might exist in the original graph, and hence (x,y) might not be an affected pair.) However, these algorithms have the advantage that they work even in the presence of zero-length cycles.

All the above-mentioned algorithms use an adaptation of Dijkstra's algorithm to solve the dynamic SSSP algorithm. The algorithms can, however, be adapted to handle negative length edges using the technique outlined in Section 6.3. The algorithms due to Rodionov [24], Murchland[20], Dionne [7], and Cheston [4], are all based on a different, and less efficient, technique of computing the new d value for every pair in A, the approximation to the set of affected pairs, using an adaptation of Floyd's algorithm for the batch shortest-path problem. A vertex pair (x,y) is considered to be possibly affected and is included in A iff $d_{old}(x,y) = d_{old}(x,u) + length_{old}(u \rightarrow v) + d_{old}(v,y)$. The adapted version of Floyd's algorithm differs from the original version in that in each of the n iterations only the d values of vertex pairs in A are recomputed. This algorithm runs in $O(|A| \cdot n)$ time.

Let us now consider the problem of updating the solution to the APSP problem after non-unit changes to the graph. This problem has not received much attention. The algorithm outlined in Section 6.2 for the dynamic APSP>0 problem is the only known incremental algorithm for any version of the dynamic APSP problem that is capable of handling insertions and deletions of edges simultaneously. Goto and Sangiovanni-Vincentelli [14] outline an incremental algorithm for updating the solution to multiple SSSP problems on the same graph when the lengths of one or more edges in the graph are decreased. Rodionov [24] considers the problem of updating the solution to the APSP problem when the lengths of one or more edges all of which have a common endpoint are decreased.

Versions of the shortest-path problem other than the all-pairs version have not received much attention either. Goto and Sangiovanni-Vincentelli [14] consider the dynamic version of the problem of solving multiple single-source shortest-path problem: given a graph G and a *set* of source vertices S, determine the length of the shortest path between s and u for every source vertex s and every vertex u. Hence, the algorithm in [14] applies to the single-source problem as a special case. The only other results concerning the dynamic SSSP problem appear in [23] and this paper.

Loubal [19], and Halder [15] study a generalization of the all-pairs shortest-path problem, where a subset S of the vertices in the graph is specified and the shortest path between any two vertices in S have to be computed.

In conclusion, the work described in this paper differs from the previous work in this area in several ways. First, the incremental algorithm we have presented is first algorithm for any version of the dynamic

shortest-path problem that is capable of handling arbitrary modifications to the graph (*i.e.*, multiple heterogeneous changes to the graph). Second, the version of the dynamic shortest-path problem we address, namely the single-source version, has been previously considered only in [14]. The algorithm described in this paper is more efficient and capable of handling more general modifications than the algorithm described in [14]. (However, the latter algorithm, unlike our algorithm, can handle negative edge lengths.) Finally, we have generalized our algorithm for a version of the dynamic fixed point problem.

Appendix

In this appendix we prove the claims made in Section 2 concerning the relationship between the various versions of the grammar problem and the various versions of the fixed point problem. We show how the SF grammar problem can be reduced to the problem of computation of the maximal fixed point of a collection of WSF equations, and how the SSF grammar problem can be reduced to the problem of computing the unique fixed point of a collection of SWSF equations. We begin by showing that the class of *w.s.f.* and *s.w.s.f.* functions are closed with respect to function composition.

Proposition A.1.

(*a*) If $g(x_1, \ldots, x_k)$ is a *s.w.s.f.* then so is the function $h(x_1, \ldots, x_m)$ defined by

$$h(x_1, \ldots, x_m) =_{def} g(x_{j_1}, \ldots, x_{j_k})$$

where every $j_i \in [1,m]$. Similarly, if g is a *w.s.f.* then so is h.

(*b*) Let $f(x_1, \ldots, x_k)$ be a *w.s.f.*, and let $g_j(x_1, \ldots, x_m)$ be a *s.w.s.f.* for every $j \in [1,k]$. The function $h(x_1, \ldots, x_m)$ defined as follows is a *s.w.s.f.* too.

$$h(x_1, \ldots, x_m) =_{def} f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$$

Further, if each g_j is a *w.s.f.*, then so is h.

Proof.

(*a*) Let g be a *s.w.s.f.* The monotonicity of g directly implies that h is monotonic. Now,

$h(x_1, \ldots, x_m) \le x_i$
$\Rightarrow\ g(x_{j_1}, \ldots, x_{j_k}) \le x_i$
$\Rightarrow\ g(x_{j_1}, \ldots, x_{j_k}) \le x_{j_p}$ for every p such that $j_p = i$
$\Rightarrow\ g(y_1, \ldots, y_k) = g(x_{j_1}, \ldots, x_{j_k})$ where $y_p =_{def}$ if $(j_p = i)$ then ∞ else x_{j_p}
using Proposition 3.1(*a*)
$\Rightarrow\ h(x_1, \ldots, \infty, \ldots, x_m) = h(x_1, \ldots, x_i, \ldots, x_m)$

It similarly follows that if g is a *w.s.f.* then h is a *w.s.f.* too.

(*b*) The monotonicity of h follows immediately from the monotonicity of f and $g_1, \ldots, g_k$. Now,

$h(x_1, \ldots, x_i, \ldots, x_k) \le x_i$
$\Rightarrow\ f(y_1, \ldots, y_k) \le x_i$ where $y_j =_{def} g_j(x_1, \ldots, x_i, \ldots, x_k)$
$\Rightarrow\ f(y_1, \ldots, y_k) < y_j$ for every $y_j > x_i$
$\Rightarrow\ f(w_1, \ldots, w_k) = f(y_1, \ldots, y_k)$ where $w_j =_{def}$ if $(y_j > x_i)$ then ∞ else y_j
(using Proposition 3.1(*a*))
$\Rightarrow\ f(w_1, \ldots, w_k) = f(y_1, \ldots, y_k)$
where $w_j =_{def}$ if $(g_j(x_1, \ldots, x_i, \ldots, x_k) > x_i)$ then ∞ else $g_j(x_1, \ldots, x_i, \ldots, x_k)$
$=$ if $(g_j(x_1, \ldots, x_i, \ldots, x_k) > x_i)$ then ∞ else $g_j(x_1, \ldots, \infty, \ldots, x_k)$
since g_j is strictly weakly superior
$\ge\ g_j(x_1, \ldots, \infty, \ldots, x_k)$
$\Rightarrow\ f(z_1, \ldots, z_k) \le f(y_1, \ldots, y_k)$ where $z_j =_{def} g_j(x_1, \ldots, \infty, \ldots, x_k)$
$\Rightarrow\ h(x_1, \ldots, \infty, \ldots, x_k) \le h(x_1, \ldots, x_i, \ldots, x_k)$
$\Rightarrow\ h(x_1, \ldots, \infty, \ldots, x_k) = h(x_1, \ldots, x_i, \ldots, x_k)$
since $h(x_1, \ldots, \infty, \ldots, x_k) \ge h(x_1, \ldots, x_i, \ldots, x_k)$ by monotonicity

The result follows. □

We now look at how the grammar problem can be reduced to the maximal fixed point problem.

Definition A.2. The collection of equations Q_G determined by an abstract grammar G consists of the following equation for each non-terminal Y in the grammar:

$$d(Y) = \min \{ g(d(X_1), \ldots, d(X_k) \mid Y \longrightarrow g(X_1, \ldots, X_{k)} \text{ is a production } \}$$

We now characterize the set of equations determined by SF and SSF grammars.

Theorem A.3. If G is a SF grammar, then Q_G is a collection of WSF equations, while if G is an SSF grammar, then Q_G is a collection of SWSF equations.

Proof. Every equation in Q_G is of the form

$$d(Y) = \min (g_1(d(X_{i_{1,1}}), \ldots, d(X_{i_{1,n(1)}})) , \ldots , g_m(d(X_{i_{m,1}}), \ldots, d(X_{i_{m,n(m)}}))).$$

Now, *min* is an *w.s.f.* It follows from Proposition A.1 that if each g_i is an *s.f.* then the above equation is an WSF equations (since a superior function is also a weakly superior function). Similarly, if each g_i is an *s.s.f.*, then the above equation is an *s.w.s.f.* (since an *s.s.f.* is also an *s.w.s.f.*). The result follows. □

We now relate the solution $m_G(Y)$ of an instance G of the grammar problem to the maximal fixed point of the collection of equations Q_G.

Lemma A.4. If G is an SF grammar then ($m_G(Y)$ | Y is a non-terminal) is a fixed point of Q_G.

Proof. First observe that if G is an SF grammar then ($m_G(Y)$ | Y is a non-terminal) is well-defined as constructively established by Knuth's algorithm to compute this collection of values.

$$\begin{aligned}
m_G(Y) &= \min_{Y \longrightarrow \cdot \alpha} val(\alpha) \quad \text{(from the definition of } m_G(Y)) \\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} \; \min_{g(X_1, \ldots, X_k) \longrightarrow \cdot \alpha} val(\alpha) \\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} \min\{val(g(\alpha_1, \ldots, \alpha_k)) \mid X_i \longrightarrow^* \alpha_i\} \\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} \min\{g(val(\alpha_1), \ldots, val(\alpha_k)) \mid X_i \longrightarrow^* \alpha_i\} \\
&\qquad \text{(from the definition of } val(g(\alpha_1, \ldots, \alpha_k))) \\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} g(\min_{X_1 \longrightarrow \cdot \alpha_1} val(\alpha_1), \ldots, \min_{X_k \longrightarrow \cdot \alpha_k} val(\alpha_k)) \quad \text{(since } g \text{ is monotonic)} \\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} g(m_G(X_1), \ldots, m_G(X_k)) \quad \text{(from the definition of } m_G)
\end{aligned}$$

□

Lemma A.5. Let G be an SF grammar, and let ($f(Y)$ | Y is a non-terminal) be a fixed point of Q_G. Then, $f(Y) \leq m_G(Y)$ for each non-terminal Y.

Proof. It is sufficient to show for every terminal string α that if Y is a non-terminal such that $Y \longrightarrow^* \alpha$, then $f(Y) \leq val(\alpha)$. The proof is by induction on the length of the string α. Assume $Y \longrightarrow^* \alpha$. Then we must have $Y \longrightarrow g(X_1, \ldots, X_k) \longrightarrow^* g(\alpha_1, \ldots, \alpha_k) = \alpha$. Since each α_i is a smaller string than α and $X_i \longrightarrow^* \alpha_i$, it follows from the inductive hypothesis that $f(X_i) \leq val(\alpha_i)$. It follows from the monotonicity of g that $g(f(X_1),...,f(X_k)) \leq g(val(\alpha_1),...,val(\alpha_k)) = val(\alpha)$. Since ($f(Y)$ | Y is a non-terminal) is a fixed point of Q we have $f(Y) \leq g(f(X_1),...,f(X_k))$. The result follows. □

Theorem A.6. Let G be an SF grammar. Then ($m_G(Y)$ | Y is a non-terminal) is the maximal fixed point of Q_G.

Proof. Immediate from lemmas A.4 and A.5. □

Theorem A.7. Let Q be a collection of k equations, the i-th equation being

$$x_i = g_i(x_1, \ldots, x_k).$$

If every g_i is an *s.w.s.f.* then Q has a unique fixed point.

Proof. The existence of a fixed point, in fact, follows from the algorithm outlined in the Section 4, which computes this fixed point. The uniqueness of the fixed point may be established as follows.

Assume, to the contrary, that $(a_i \mid 1 \le i \le k)$ and $(b_i \mid 1 \le i \le k)$ are two different fixed points of Q. Choose the least element of the set $\{ a_i \mid a_i \ne b_i \} \cup \{ b_i \mid a_i \ne b_i \}$. Without loss of generality, assume that the least element is a_i. Thus, we have $a_i < b_i$, and also $a_j = b_j$ for all $a_j < a_i$. Now, we derive a contradiction as follows.

$$\begin{aligned}
a_i &= g_i(a_1, \ldots, a_k) && \text{since } (a_i \mid 1 \le i \le k) \text{ is a fixed point of } Q \\
&= g_i(c_1, \ldots, c_k) && \text{where } c_j =_{def} \text{ if } (a_j < a_i) \text{ then } a_j \text{ else } \infty \\
&&& \text{(since } g_i \text{ is a strict } w.s.f.\text{)} \\
&= g_i(c_1, \ldots, c_k) && \text{where } c_j =_{def} \text{ if } (a_j < a_i) \text{ then } b_j \text{ else } \infty \\
&&& \text{(since } a_j = b_j \text{ whenever } a_j < a_i\text{)} \\
&\ge g_i(b_1, \ldots, b_k) && \text{since } c_j \ge b_j \text{ for every } j \in [1,k] \\
&\ge b_i && \text{since } (b_i \mid 1 \le i \le k) \text{ is a fixed point of } Q.
\end{aligned}$$

The contradiction implies that Q has a unique fixed point. □

We now summarize the above results. Theorems A.3 and A.6 show how the SF grammar problem can be reduced to the WSF maximal fixed point problem. Theorems A.3, A.6, and A.7 establish that the SSF grammar problem can be reduced to the SWSF fixed point problem.

It is worth mentioning at this point that the above results hold in somewhat more general form. Define a WSF grammar to be an abstract grammar in which every production function is a *w.s.f.*, and an SWSF grammar to be an abstract grammar in which every production function is an *s.w.s.f.* The grammar problem for a WSF grammar that has no useless symbols—a context-free grammar is said to have no useless symbols if each non-terminal in the grammar can derive at least one terminal string—can be solved by reducing it to the WSF maximal fixed point problem. It is straightforward to show that, conversely, the WSF maximal fixed point problem can be reduced to the grammar problem for a WSF grammar with no useless symbols. Similarly, the grammar problem for an SWSF grammar with no useless symbols is equivalent to the SWSF fixed point problem.

References

1. Ausiello, G., Italiano, G.F., Spaccamela, A.M., and Nanni, U., "Incremental algorithms for minimal length paths," pp. 12-21 in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, (San Francisco, CA, Jan. 22-24, 1990), Society for Industrial and Applied Mathematics, Philadelphia, PA (1990).
2. Ausiello, G., Italiano, G.F., Spaccamela, A.M., and Nanni, U., "Incremental algorithms for minimal length paths," *Journal of Algorithms*, (12) pp. 615-638 (1991).
3. Carroll, M.D., "Data flow update via dominator and attribute updates," Ph.D. dissertation, Rutgers University, New Brunswick, NJ (May 1988).
4. Cheston, G.A., "Incremental algorithms in graph theory," Ph.D. dissertation and Tech. Rep. 91, Dept. of Computer Science, University of Toronto, Toronto, Canada (March 1976).
5. Cheston, G.A. and Corneil, D.G., "Graph property update algorithms and their application to distance matrices," *INFOR* **20**(3) pp. 178-201 (August 1982).
6. Dijkstra, E.W., "A note on two problems in connexion with graphs," *Numerische Mathematik* **1** pp. 269-271 (1959).
7. Dionne, R., "Etude et extension d'un algorithme de Murchland," *INFOR* **16**(2) pp. 132-146 (June 1978).
8. Driscoll, J.R., Gabow, H.N., Shrairman, R., and Tarjan, R.E., "Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation," *Communications of the ACM* **31**(11) pp. 1343-1354 (1988).

9. Edmonds, J. and Karp, R.M., "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM* **19** pp. 248-264 (1972).

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

11. Fredman, M.L. and Tarjan, R.E., "Fibonacci heaps and their uses in improved network optimization algorithms," *J. ACM* **34**(3) pp. 596-615 (1987).

12. Fredman, M.L. and Willard, D.E., "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," pp. 719-725 in *Proceedings of the 31st Annual Symposium on Foundations of Computer Science Volume II* (St. Louis, Missouri, October 1990), IEEE Computer Society, Washington, DC (1990).

13. Gondran, M. and Minoux, M., *Graphs and Algorithms*, John Wiley and Sons, New York (1984).

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

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

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

17. Knuth, D.E., "A generalization of Dijkstra's algorithm," *Information Processing Letters* **6**(1) pp. 1-5 (1977).

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

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

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

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

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

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

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

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

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

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

28. Yap, C.K., "A hybrid algorithm for the shortest path between two nodes in the presence of few negative arcs," *Information Processing Letters* **16** pp. 181-182 (May 1983).