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

G. Ramalingam and Thomas Reps
University of Wisconsin – Madison

*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

Incremental Shortest-Path Problem

G. Ramalingam
Computer Sciences Department
University of Wisconsin—Madison
1210 W. Dayton St.
Madison, WI 53706

ABSTRACT

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 each non-terminal of a given context-free grammar, with the cost of a derivation being suitably defined. This problem also subsumes the problem of finding optimal hyperpaths in directed hypergraphs (under varying optimization criteria) that has received attention recently. 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 work that distinguishes it from 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.

1. Introduction

Knuth defined the following generalization of the single-source shortest-path problem, called the *grammar problem* [24]: 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 each non-terminal of the given grammar.

Knuth showed that it is possible to adapt Dijkstra's shortest-path algorithm [11] 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.

The grammar problem also subsumes a number of hypergraph problems [3, 6, 7, 18, 23], in particular, the problem of finding optimal hyperpaths in directed hypergraphs [5]. A directed hypergraph consists of a set of nodes and a set of hyperarcs, where each hyperarc connects a set of source nodes to a single target node. A hypergraph corresponds to a context-free grammar, nodes correspond to non-terminals, and hyperarcs correspond to productions. A *hyperpath* in a hypergraph corresponds to a *derivation* in the grammar. The concept of a value-based measure for a hyperpath that Ausiello *et al.* introduce in [5] is similar to the cost assigned to a derivation in Knuth's grammar problem, and an optimal hyperpath corresponds to a minimum-cost derivation. The correspondence between hypergraph problems and grammar problems is discussed in detail in Section 6, but it is worth mentioning here that the grammar problem is a more general problem because it allows more general kinds of cost functions to be used than the ones permitted with value-based measures.

The main contributions of this paper are:

- Knuth identified a class of grammar problems that can be solved by an algorithm that is essentially Dijkstra's algorithm. We identify a larger class of problems that can be solved in this way.
- We present an algorithm for a class of dynamic grammar problems. A variant of Knuth's algorithm for the batch grammar problem is obtained as a special case of our algorithm for the dynamic grammar problems (when a collection of productions is "inserted" into an empty grammar). However, our incremental algorithm encounters "configurations" that can never occur in any run of the batch algorithm.
- As an important 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*). Again, Dijkstra's algorithm turns out to be a special case of our algorithm (when a collection of edges is inserted into an empty graph).
- Utilizing our incremental algorithm for the dynamic SSSP>0 problem, we present an improved algorithm for the batch single-pair (and single-source) shortest path problem in graphs with a small number of negative edges.

The aspect of our work that distinguishes it from 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 *unit changes*—*i.e*, exactly one edge is inserted, deleted, or changed in length.

---

[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.

In general, a single application of an algorithm for heterogeneous changes has the potential to perform significantly better than the repeated application of an algorithm for unit 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 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 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 related to “the size of the change in the input and output”. Specifically, for the dynamic SSSP>0 problem, $\|\delta\|$ is the sum of the number of vertices whose values change and the number of edges incident on these vertices. Our dynamic SSSP>0 algorithm updates a graph, after an arbitrary mixture of edge insertions, edge deletions, and edge-length changes, in time $O(\|\delta\| \log \|\delta\|)$. Our dynamic algorithm for the grammar problem runs in time $O(\|\delta\|(log\|\delta\| + M))$, where M is a bound on the time required to compute any production function and $\|\delta\|$, as before, is a measure of the size of the change in the input and output (except that, now, it is the sum of the number of non-terminals whose values change and the number of productions associated with these non-terminals). (For a formal definition of $\|\delta\|$ and the notion of boundedness, see Section 2.)

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 first develop the idea behind the algorithm via a sequence of lemmas about the problem. We then present the first version of our algorithm, a proof of its correctness, and an analysis of its time complexity. In Section 4, we discuss an improved version of the algorithm, and analyze its time complexity. In Section 5, we look at some extensions of the algorithm. In Section 6, we discuss related work. The paper ends with an appendix that covers some relevant results and their proofs.

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 and the Grammar 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 5.3, 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. We refer to these two problems as the SSSP≥0 problem and SSSP>0 problem, respectively.

The grammar problem is a generalization of the shortest path problem due to Knuth [24]. In the rest of the paper let $(D, \leq, \infty)$ be a totally ordered set with maximum element ∞.[2] We define an ***abstract grammar*** (with value domain 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 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 shortly, guarantee that $m_G(Y)$ is well defined.

Examples. (Knuth [24].) 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

[2] 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.

immaterial since the parentheses were used in the definition of the grammar problem just as a notational convenience.) Note that a grammar problem of this form corresponds to an SSSP problem only if it 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". □

We now consider certain classes of grammar problems obtained by placing some restrictions on the production functions. 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. Knuth shows how Dijsktra's algorithm for computing shortest paths can be generalized to solve the grammar problem for the class of SF grammars.

It turns out that the Dijkstra/Knuth algorithm can be adapted to solve a larger class of grammar problems than the class of SF grammar problems. We first define two classes of functions that generalize the class of superior and strict superior functions, respectively. 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 non-nullary constant-valued function is another example of an *s.w.s.f.* that is not an *s.s.f.*

The Dijsktra/Knuth algorithm can be adapted to solve all (batch) *w.s.f.* grammar problems, as follows: First, all useless nonterminals—and productions containing useless nonterminals—are removed. (A useless non-terminal is one that cannot derive any terminal string. The set of useless non-terminals in a grammar can be found in time linear in the size of the grammar. The problem of identifying useless non-terminals can also be expressed as a grammar problem.) Second, the Dijkstra/Knuth algorithm is applied to the grammar problem that remains.

The Fixed Point Problem

The various grammar problems defined above can also be related to certain fixed-point problems. For example, every input instance G of SSSP induces a collection of equations, called the *Bellman-Ford* equations, in the set of unknowns $\{ d(u) \mid u \in V(G) \}$:

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

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 [19] 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. Furthermore, 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.

Similarly, every instance of the grammar problem corresponds to a collection of mutually recursive equations. Each non-terminal Y in the grammar gives rise to the following equation:

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

The motivation behind the definition of *w.s.f.* and *s.w.s.f.* functions is the following observation:

> The right-hand side of the above equation, in general, is a function of the form $\min(g_1(x_1, \ldots, x_k), \ldots, g_m(x_1, \ldots, x_k))$. *This function is not, in general, a s.f. function even when g_1 through g_m are s.f. functions.* However, as shown in the Appendix (Proposition A.1(b)), this function *is* a *w.s.f.* function, as long as g_1 through g_m are *w.s.f.* (or *s.f.*) functions. Furthermore, if g_1 through g_m are *s.w.s.f.*, then the above function is also *s.w.s.f.*

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

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

We say that an equation of the above form is a ***WSF equation*** if g_i is a *w.s.f.*, and an ***SWSF equation*** if g_i is an *s.w.s.f.* One consequence of the above observation is:

> The set of equations (*) arising from Knuth's SF and SSF grammar problems *are not*, in general, SF and SSF equations; however, they *are* WSF and SWSF equations.

We define the ***WSF maximal fixed point problem*** to be that of computing the maximal fixed point of a collection of WSF equations. We show in the Appendix that the solution of a WSF (or SF) grammar problem may be obtained by solving the corresponding WSF maximal fixed point problem (*i.e.*, equations (*)), provided that the grammar has no useless non-terminals (see Theorems A.3 and A.6).

A collection of SWSF equations can be shown to have a unique fixed point (see Theorem A.7). 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 SWSF (and SSF) grammar problem, since, as we show in the Appendix, each equation in the collection of equations determined by an SWSF grammar is an SWSF equation (see Theorems A.3 and A.6). The SSSP>0 problem is obtained as yet a further special case of the SWSF 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 instance G of a 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+δ. A vertex u of G+δ is said to be a ***modified*** vertex if the equation associated with u was changed. The set of all modified vertices in G+δ will be denoted by $Modified_{G,\delta}$. This set captures the change in the input.

A vertex in G+δ is said to be an ***affected*** vertex if its output value in G+δ is different from its output value in G. Let $Affected_{G,\delta}$ denote the set of all affected vertices in G+δ. 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 a parameter related to 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 [1, 32, 33].

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 a fixed point 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 $|G|$). 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 $M_{G,\delta}$ (abbreviated M_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 $M_{G,\delta}$. The algorithms presented in Sections 3 and 4 are both bounded scheduling cost algorithms. The algorithm presented in Section 4 is a bounded incremental algorithm for the special case of the dynamic SSSP>0 problem.

Remark. The use of $\|\delta\|$, as opposed to $|\delta|$, in describing the complexity of an algorithm indicates that the behavior of that algorithm depends on the degree to which the set of vertices whose values change are connected to vertices with unchanged values. Some readers have objected that $\|\delta\|$ hides a dependency of the running time of our algorithms on the input size. We believe this to be an unjust criticism, for the following reason:

- The parameter $\|\delta\|$ is an *adaptive* parameter, one that varies from 1 to $|E(G)| + |V(G)|$. This is similar to the use of adaptive parameter $|E(G)| + |V(G)|$—which ranges from $|V(G)|$ to $|V(G)|^2$—to describe the running time of depth-first search. Note that if allowed to use only the parameter $|V(G)|$, one would have to express the complexity of depth-first search as $O(|V(G)|^2)$—which provides less information than the usual description of depth-first search as an $O(|E(G)| + |V(G)|)$ algorithm. For instance, when the behavior of depth-first search is characterized as $O(|E(G)| + |V(G)|)$, one can infer that the complexity of depth-first search on bounded-degree graphs is $O(|V(G)|)$ (and not $O(|V(G)|^2)$). Similarly, when the behavior of our incremental shortest path algorithm is characterized as $O(||D|| \log ||D||)$, one can infer that the algorithm's complexity on bounded-degree graphs is $O(|D| \; |D|)$ (and not $O(|V(G)| \times |D| \times \log |V(G)|)$).

The quantity $\|\delta\|$ is merely a name for a natural parameter that arises in the kind of dynamic path problems we are considering: the total indegree of the vertices whose values change. □

3. An Algorithm for the Dynamic SWSF Fixed Point Problem

The dynamic SWSF fixed point problem is to maintain the unique fixed point of a collection of SWSF equations as they undergo changes. In this section, we present an algorithm for the problem. An outline of the algorithm is presented as procedure *IncrementalFP* in Figure 2 and discussed in Section 3.1. The full algorithm is presented as procedure *DynamicSWSF−FP* in Figure 3 and discussed in Section 3.2.

3.1. The Idea Behind the Algorithm

Assume that the given collection of equations consists of k equations in the k unknowns x_1 through x_k, the i-th equation being $x_i = g_i(x_1, \ldots, x_k)$. For convenience, we will refer to the function associated with a vertex (variable) x_i by both g_i and g_{x_i}. Every vertex x_i has an associated tentative output value $d[x_i]$, which denotes the value of x_i in the unique fixed point of the collection of equations before modification. Thus, it is the previous output value of vertex x_i. (We use square brackets, as in $d[x_i]$, to indicate variables whose values are maintained by the program.) Let $d^*(x_i)$ denote the actual output value that vertex x_i should have in the unique fixed point of the modified collection of equations. 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 $g_i(d[x_1], \ldots, d[x_k])$—it denotes the value of the right-hand side of the equation associated with the variable x_i under the given assignment of values to variables. We say that vertex x_i is ***consistent*** if

$$d[x_i] = rhs(x_i).$$

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

$$d[x_i] > rhs(x_i).$$

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

$$d[x_i] < rhs(x_i).$$

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 $d[u] < d^*(u)$. Note, however, that 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".

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 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 as well. In other words, in the dynamic SSSP>0 algorithm, a vertex u can, at some stage, have a distance $d[u]$ that is *strictly less than* $\min_{v \in Pred(u)} [d[v] + length(v \to u)]$. This situation never occurs in Dijkstra's algorithm.

Under-consistent vertices can arise in the dynamic shortest-path problem, for instance, when some edge on some shortest path is deleted. This introduces some complications. 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 over-consistent vertex is guaranteed to be an over-estimated vertex.) See Figure 1 for an example illustrating this.

If we change the value of an inconsistent but correct vertex to make it consistent, we may end up with an unbounded algorithm, which leaves us with two questions:

What is the right order for processing inconsistent vertices?

We will show 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:

$$key(x_i) =_{def} min(\ d[x_i],\ \ rhs(x_i)\ ).$$

In other words, the key of an over-consistent vertex x_i is $rhs(x_i)$, while the key of an under-consistent vertex x_i is $d[x_i]$. As we will soon show, 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.

How is an inconsistent vertex to be processed?

We will show that if the inconsistent vertex with the least key is over-consistent, then its *rhs* value is

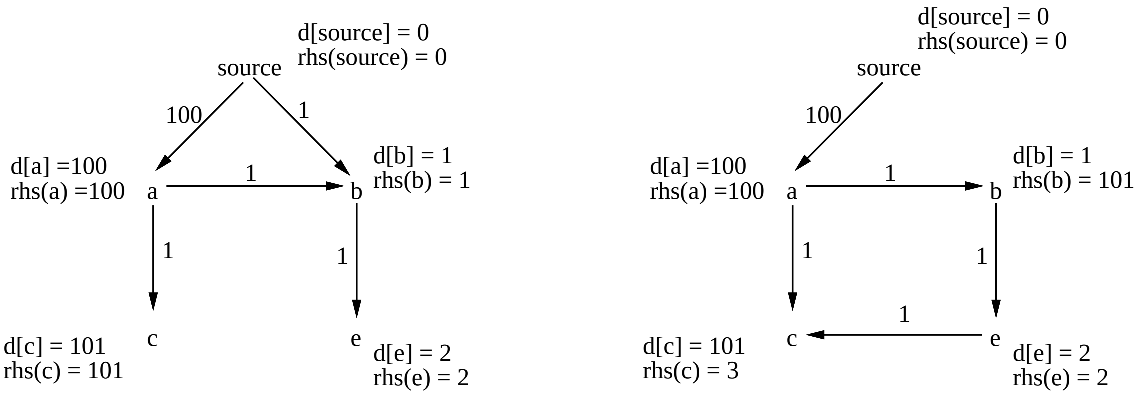

Figure 1. Example of over-consistent and under-consistent vertices in the dynamic SSSP>0 problem. The figure on the left indicates a graph for which the single-source shortest-path information has been computed. All vertices are consistent in this graph. The simultaneous deletion of the edge $source \rightarrow b$ and the insertion of the edge $e \rightarrow c$ make vertex b under-consistent and vertex c over-consistent. Observe that though c is inconsistent its value is correct.

its correct value. 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 now present an outline of the algorithm in Figure 2. The algorithm works by repeatedly selecting an inconsistent variable whose *key* is less than or equal to the key of every other inconsistent variable and processing it. If the selected variable u is under-consistent, then it is assigned a new value of ∞, and if it is over-consistent, then it is assigned the new value of $rhs(u)$.

We now wish to establish three properties: (a) that the algorithm is correct, (b) that the algorithm does not change the value of any unaffected vertex, and (c) that the algorithm changes the value of a vertex at most twice. These results follow once we establish the following two claims:

```
procedure IncrementalFP (Q)
declare
        Q : a set of SWSF equations
        rhs(u) =def g_u(d[x_1], . . . , d[x_k])
        key(u) =def min(d[u], rhs(u))
begin
        while there exist inconsistent variables in Q do
           let u be an inconsistent vertex with minimum key value
           if rhs[u] < d(u) then        // u is over-consistent
              d[u] := rhs(u)
           else if d[u] < rhs(u) then   // u is under-consistent
              d[u] := ∞
           fi
        od
end
```

Figure 2. An algorithm to update the unique fixed point of a collection of SWSF equations after a change in the collection of equations. Note that *rhs* and *key* are functions.

Claim (1):

If the vertex u chosen in line [2] is assigned a value in line [4] (in some particular iteration), then vertex u becomes consistent and remains consistent subsequently.

Claim (2):

If the vertex u chosen in line [2] is assigned a value in line [6] (in some particular iteration), then u will never subsequently be assigned the same value it had before the execution of line [6].

The correctness of the algorithm will follow from Claim (1): it follows from the claim that line [4] can be executed at most once for each vertex u; line [6], too, can be executed at most once for each vertex u, since once $d[u]$ is set to ∞, u cannot subsequently become an under-consistent vertex—as long as $d[u]$ is ∞, it satisfies the condition in line [3]; hence $d[u]$ can only be changed subsequently in line [4], in which case the vertex becomes consistent and remains so, from Claim (1). Hence, the algorithm performs at most two iterations for each vertex, and consequently the algorithm must halt. The correctness follows immediately from the termination condition for the loop.

How can we establish that the algorithm is bounded? Claims (1) and (2) imply that the values of only affected variables are changed by the algorithm, and it follows from Claim (1) that the algorithm performs at most two iterations for each affected variable. It follows that the algorithm performs a bounded number of iterations. We will show later that line [2] can be implemented to run in bounded time, which suffices to establish that the algorithm is bounded.

Before proving Claims (1) and (2), we need to first establish some properties of *s.w.s.f.* functions. 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.* and $g(x_1, \ldots, x_k) < g(y_1, \ldots, y_k)$ then there exists $i \in [1,k]$ such that $x_i < g(x_1, \ldots, x_k)$ and $x_i < y_i$.

(d) If g is a *s.w.s.f.* and $g(x_1, \ldots, x_i, \ldots, x_k) \neq g(x_1, \ldots, x_i', \ldots, x_k)$, then $g(x_1, \ldots, x_i, \ldots, x_k) > min(x_i, x_i')$ and, similarly, $g(x_1, \ldots, x_i', \ldots, x_k) > min(x_i, x_i')$.

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.

$$
\begin{array}{lll}
g(x_1, \ldots, y, \ldots, x_k) \neq g(x_1, \ldots, x_i, \ldots, x_k) & & \\
\quad \Rightarrow\; g(x_1, \ldots, y, \ldots, x_k) \neq g(x_1, \ldots, \infty, \ldots, x_k) & & \text{(since } g \text{ is an } s.w.s.f. \\
\quad \Rightarrow\; g(x_1, \ldots, y, \ldots, x_k) < g(x_1, \ldots, \infty, \ldots, x_k) & & \text{(since } g \text{ is monotonic)} \\
\quad \Rightarrow\; g(x_1, \ldots, y, \ldots, x_k) < g(x_1, \ldots, x_i, \ldots, x_k) & & \text{(since } g \text{ is an } s.w.s.f.) \\
\quad \Rightarrow\; g(x_1, \ldots, y, \ldots, x_k) < y & & \text{(from assumption about } y) \\
\quad \Rightarrow\; g(x_1, \ldots, y, \ldots, x_k) = g(x_1, \ldots, \infty, \ldots, x_k) & & \text{(since } g \text{ is an } s.w.s.f.) \\
\quad \Rightarrow\; g(x_1, \ldots, y, \ldots, x_k) = g(x_1, \ldots, x_i, \ldots, x_k) & & \text{(since } g \text{ is an } s.w.s.f.)
\end{array}
$$

This proves (1); (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,

$$g(x_1, \ldots, x_k) \;=\; g(z_1, \ldots, z_k)$$

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

The result follows.

(d) This follows directly from (b), since if $g(x_1, \ldots, x_i, \ldots, x_k) \leq x_i$, then $g(x_1, \ldots, x_i, \ldots, x_k) = g(x_1, \ldots, y, \ldots, x_k)$ for all $y \geq g(x_1, \ldots, x_i, \ldots, x_k)$. □

We will now prove Claims (1) and (2). These claims follow from the fact that the keys of vertices chosen in line [2] over the iterations form a non-decreasing sequence. We first consider the change in the consistency status and the key value of vertices when a vertex u is processed (lines [3]-[7]) in some particular iteration. Let us denote the "initial" values of variables and expressions, that is, the value of these variables and expressions before the execution of lines [3]-[7] in the iteration under consideration, with the subscript "old", and to the "final" values of these variables and expressions with the subscript "new". In the following propositions u denotes the vertex chosen in line [2] of the particular iteration under consideration.

Proposition 3.2. If $rhs_{new}(w) \neq rhs_{old}(w)$ then $rhs_{new}(w) > key_{old}(u)$ and $rhs_{old}(w) > key_{old}(u)$.

Proof. Note that $rhs_{new}(w) = g_w(d[x_1], \ldots, d_{new}[u], \ldots, d[x_k])$, while $rhs_{old}(w) = g_w(d[x_1], \ldots, d_{old}[u], \ldots, d[x_k])$. It follows from Proposition 3.1(d) that both $rhs_{old}(w)$ and $rhs_{new}(w)$ are greater than $min(d_{old}[u], d_{new}[u]) = min(d_{old}[u], rhs_{old}(u)) = key_{old}(u)$. □

Proposition 3.3. If u is over-consistent at the beginning of the iteration, then it is consistent at the end of the iteration.

Proof. Since u is over-consistent initially, $d[u]$ will be assigned the value $rhs_{old}(u)$. As long as this assignment does not change the *rhs* value of u, u must be consistent. But $rhs_{old}(u)$ must be equal to $rhs_{new}(u)$—since, otherwise, we would have $rhs_{old}(u) > key_{old}(u) = rhs_{old}(u)$, which is a contradiction. □

Proposition 3.4. For any vertex w that is inconsistent at the end of the iteration, $key_{new}(w) \geq key_{old}(u)$.

Proof. Since $key_{new}(w) = min(rhs_{new}(w), d_{new}[w])$, by definition, we need to show that (a) $rhs_{new}(w) \geq key_{old}(u)$, and that (b) $d_{new}[w] \geq key_{old}(u)$. Consider (a). If the *rhs* value of w did not change, then w must have been inconsistent originally. Hence, $rhs_{new}(w) = rhs_{old}(w) \geq key_{old}(w) \geq key_{old}(u)$. If the *rhs* value of w did change, then it follows from Proposition 3.2 that $rhs_{new}(w) \geq key_{old}(u)$. Now consider (b). If w was originally inconsistent, then $d_{new}[w] \geq d_{old}[w] \geq key_{old}(w) \geq key_{old}(u)$. If w was originally consistent, then $rhs(w)$ must have changed in value. It follows from Proposition 3.2 that $d_{new}[w] = d_{old}[w] = rhs_{old}(w) > key_{old}(u)$. □

We now turn our attention to how the values and consistency statuses of variables change over the different iterations of the algorithm. The subscript i attached to any variable or expression denotes the value of the variable or expression at the beginning of iteration i.

Proposition 3.5. If $i < j$ then $key_i(u_i) \leq key_j(u_j)$. In other words, the keys of variables chosen in line [2] form a monotonically non-decreasing sequence.

Proof. This follows trivially from repeated applications of Proposition 3.4. □

Proposition 3.6. Assume that the vertex u_i chosen in line [2] of the i-th iteration is an over-consistent vertex. Then, u_i remains consistent in all subsequent iterations. In particular, its value is never again changed.

Proof. We showed above in Proposition 3.3 that variable u_i is consistent at the end of the i-th iteration. It can never again become inconsistent because its *rhs* value can never again change—this follows because, if $rhs(u_i)$ were to change in a subsequent iteration, say the j-th iteration, then we would have $key_i(u_i) = rhs_i(u_i) = rhs_j(u_i) > key_j(u_j)$, from Proposition 3.2. But this contradicts Proposition 3.5. Since only the values of inconsistent variables are ever changed it follows that $d[u_i]$ is never again changed. □

Proposition 3.7. Assume that the vertex u_i chosen in line [2] of the i-th iteration is an under-consistent vertex. Then, u_i is never assigned its original value, $d_i[u_i]$, again.

Proof. We need to show that the *rhs* value of variable u_i never becomes $d_i[u_i]$. This follows from Proposition 3.2 since if $rhs(u_i)$ changes in the j-th iteration, then we have $rhs_{j+1}(u_i) > key_j(u_j) \geq key_i(u_i) = d_i[u_i]$. □

Proposition 3.8. Procedure *IncrementalFP* correctly computes the unique fixed point of the given collection of equations. Furthermore, during the course of the computation it changes only the values of the variables that had an incorrect value at the beginning of the update. It also changes the value of a variable at most twice.

Proof. This proposition follows directly from Proposition 3.6 and Proposition 3.7, as explained earlier. □

3.2. The Algorithm

We now present a detailed version of the algorithm. The algorithm is described as procedure *DynamicSWSF−FP* in Figure 3. 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, vertices not in U are guaranteed to

```
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
        Heap := ∅
        for u ∈ U do
            rhs[u] := g_u(d[x_1], . . . , d[x_k])
            if rhs[u] ≠ d[u] then
                InsertIntoHeap( Heap, u, min(rhs[u], d[u]))
            fi
        od
        while Heap ≠ ∅ do
            u := ExtractAndDeleteMin( Heap )
            if rhs[u] < d[u] then /* u is over-consistent */
                d[u] := rhs[u]
                for v ∈ Succ(u) do
                    rhs[v] := g_v(d[x_1], . . . , d[x_k])
                    if rhs[v] ≠ d[v] then
                        AdjustHeap(Heap, v, min(rhs[v], d[v]))
                    else
                        if v ∈ Heap then Remove v from Heap fi
                    fi
                od
            else /* u is under-consistent */
                d[u] := ∞
                for v ∈ (Succ(u) ∪ {u}) do
                    rhs[v] := g_v(d[x_1], . . . , d[x_k])
                    if rhs[v] ≠ d[v] then
                        AdjustHeap(Heap, v, min(rhs[v], d[v]))
                    else
                        if v ∈ Heap then Remove v from Heap fi
                    fi
                od
            fi
        od
end
postconditions
        Every vertex in V(G) is consistent
```

Figure 3. An algorithm for the dynamic SWSF fixed point problem.

be consistent. This is the precondition for the algorithm to compute the correct solution to the modified set of equations.

In order to implement line [2] of procedure IncrementalFP efficiently, we maintain a heap of all the inconsistent vertices in the graph. In order to identify the key of inconsistent vertices, the algorithm also maintains $rhs[u]$, the value of the right-hand side of the equation associated with vertex u, for every inconsistent vertex u. We briefly explain what each heap operation does. The operation $InsertIntoHeap(H,i,k)$ inserts an item i into heap H with a key k. The operation $FindAndDeleteMin(H)$ returns the item in heap H that has the minimum key and deletes it from the heap. The operation $AdjustHeap(H,i,k)$ inserts an item i into $Heap$ with key k if i is not in $Heap$, and changes the key of item i in $Heap$ to k if i is in $Heap$.

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.

The loop in lines [8]-[31] processes the inconsistent vertices as explained earlier. In addition, the algorithm also has to identify the change in consistency status of vertices and changes in the keys of inconsistent vertices as a result of the assignment of a new value to $d[u]$ in line [11] or line [21]. This is done in lines [12]-[19] and lines [22]-[29].

Let us now determine the time complexity of the algorithm. Let M_δ be 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 $O(|U| \cdot (M_\delta + \log |U|))$ time, which is $O(|\delta| \cdot (M_\delta + \log |\delta|))$ 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 $O(\|\{u\}\| \cdot (M_\delta + \log \|\delta\|))$ 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 $O(\|\{u\}\| \cdot (M_\delta + \log \|\delta\|))$ time (in one iteration).

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

4. An Improved Algorithm for the Dynamic SSF Grammar Problem

The algorithm presented in the previous section can be improved further in the case of the dynamic grammar problem. The sources of optimization are lines [13] and [23] in Figure 3. In the general SWSF fixed point problem, the function on the right-hand side of an equation can be an arbitrary *s.w.s.f.*, and, consequently, the right-hand side has to computed from "scratch" (in lines [13] and [23]) when one of the arguments changes in value. The functions that arise in the grammar problem, however, have a special form. For example, in the shortest-path problem 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. However, maintaining the whole set as a heap results in some unnecesary computation, and 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,

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

```
procedure DynamicSSF-G (G, P)
declare
        G : a SSF grammar;
        P : the set of modified productions in G
        GlobalHeap: a heap of non-terminals
        Heap: array[Nonterminals] of heap of productions;
        SP: array[Nonterminals] of set of productions
preconditions:  Every production in G−P is consistent. (See Definition 4.1)
        procedure recomputeProductionValue(p : a production)
        begin
          let p be the production Y ⟶ g(X1, . . . , Xk)
          value = g(d[X1], . . . , d[Xk])
          if (value < d[Y]) then
             AdjustHeap( Heap[Y], p, value)
          else
             if p ∈ Heap[Y] then Remove p from Heap[Y] fi
          fi
          if (value ≤ d[Y]) then SP[Y] := SP[Y] ∪ {p} else SP[Y] := SP[Y]−{p} fi
          if (SP[Y] = ∅) then     /* Y is under-consistent */
             AdjustHeap( GlobalHeap, Y, d[Y])
          elseif Heap[Y] ≠ ∅ then     /* Y is over-consistent */
             AdjustHeap( GlobalHeap, Y, min−key(Heap[Y]))
          else   /* Y is consistent */
             if Y ∈ GlobalHeap then Remove Y from GlobalHeap fi
          fi
        end
begin
        GlobalHeap := ∅
        for every production p ∈ P do
            recomputeProductionValue(p)
        od
        while GlobalHeap ≠ ∅ do
            Select and remove from GlobalHeap a non-terminal X with minimum key value
            if key(X) < d[X] then /* X is over-consistent */
                d[X] := key(X)
                SP[X] := { p | p is a production for X such that value(p) = d[X] }
                Heap[X] := ∅
                for every production p with X on the right-hand side do recomputeProductionValue(p) od
            else /* X is under-consistent */
                d[X] := ∞
                SP[X] := { p | p is a production for X }
                Heap[X] := makeHeap({ p | p is a production for X with value(p) < d[X] })
                if Heap[X] ≠ ∅ then AdjustHeap( GlobalHeap, X, min−key(Heap[X])) fi
                for every production p with X on the right-hand side do recomputeProductionValue(p) od
            fi
        od
end
postconditions:  Every non-terminal and production in G is consistent
```

Figure 4. An algorithm for the dynamic SSF grammar problem.

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]-[34] 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 is 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 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 $d[X]$. 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 $value(p) \le d[Y]$ 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 $d[X] < \infty$,[3] 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 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 4.1. 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, $value(p) \ge d[Y]$.

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 computing 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].

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.

---

[3]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.

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 Algorithm *DynamicSWSF−FP* the individual equations were treated as indivisible units, the smallest units 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 $\mathit{Affected} \cup \mathit{Modified}$.

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 $2 \cdot |\mathit{Affected}|$ 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) $O(\|\mathit{Changed}\|_B)$ invocations of the procedure *recomputeProductionValue* (lines [18], [27] and [35]), (b) $O(|\mathit{Affected}|)$ operations on *GlobalHeap* (line [21]), and (c) the remaining steps, which take time $O(\|\mathit{Changed}\|_B)$.

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) $O(1)$ $\mathit{Heap}[Y]$ operations (lines [4] or [6]), and (d) $O(1)$ *GlobalHeap* operations (lines [10], [12] or [14]). The set operations on $\mathit{SP}[Y]$ 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 $\mathit{SP}[Y]$ or not. It can be easily verified that each $\mathit{Heap}[Y]$ and *GlobalHeap* have at most $\|\mathit{Affected}\|_B$ elements. Consequently, each heap operation takes at most $\log\|\mathit{Affected}\|_B$ time.

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

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 [13] or Fibonacci heaps [16] 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 [16]. The asymptotic complexity of the algorithm can be further improved by using the recently developed AF-heap data structure [17].

5. Extensions to the Algorithms

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

5.1. Answering Queries on Demand

The incremental algorithms presented in this paper update the solution to the whole problem instance when they are invoked. This can potentially result in unnecessary computation being performed, since (potentially large) parts of the computed solution may never be used before they are “affected” by subsequent modifications to the input. In such situations it may be appropriate to use a demand-driven algorithm, where the solution to the problem instance is computed as and when necessary.

Procedures *DynamicSSF−G* and *DynamicSWSF−FP* can be easily adapted to work in such a demand-drived fashion. Consider the update-query model of dynamic algorithms commonly used: assume that the algorithm has to process a sequence of requests, where each request is either an update to the input grammer (set of equations, in the case of a fixed point problem) or a query asking for the cost of the optimal derivation from a specific non-terminal (value of a variable). For each update operation, the algorithm does nothing more than “note” down the actual modification performed. When a query is performed, the algorithm starts processing the sequence of updates performed since the last time a query was processed by invoking *DynamicSSF−G* (*DynamicSWSF−FP*). However, instead of running the algorithm to completion, we may stop the processing when we know the correct answer to the specific query being processed. It follows from the discussion in Section 3 (see, in particular, Proposition 3.2 and the proof of Proposition 3.6) that any consistent vertex whose value is less than *key(u)*, where u is the inconsistent vertex with the minimum key value, is, in fact, correct. Consequently, in this version of the dynamic SSF grammar problem, for example, if the query asks for the value $d[Y]$ of some non-terminal Y, then the updating algorithm in Figure 4 may stop when the key of the non-terminal X selected in line [21] is greater than $d[Y]$.

Some other minor modifications to the algorithms are necessary. For example, in Figure 4, *GlobalHeap*, the heap of inconsistent vertices, need no longer be empty in between updates or queries. Consequently, the algorithm should no longer initialize *GlobalHeap* to be empty at the beginning of the update, but just carry the value over from the previous invocation of the algorithm.

5.2. The Batch WSF Grammar and Fixed Point Problems

As mentioned earlier, we obtain the Knuth-Dijkstra algorithm for the batch grammar problem as a special case of procedure *DynamicSSF−G*, the dynamic algorithm presented in Figure 4. In particular, we can solve the batch problem for an input instance G as follows: assume that we initially have an empty grammar (with no productions and the same set of non-terminals as G); the solution to this empty grammar is trivial, as the output value for every non-terminal has to be ∞; we now insert all the productions of G into this empty grammar, and invoke the incremental algorithm. This yields the solution to the batch problem.

Procedure *DynamicSSF−G* was specifically for SSF grammars. However, the same algorithm works correctly even for SF grammars, as long as the input modifications consist only of the insertion of new productions, and not the deletion of productions. In particular, for the kind of batch problem addressed in the previous paragraph, the algorithm works correctly for SF grammars, too. Procedure *DynamicSSF−G* works correctly even for SWSF grammars (and WSF grammars, if input modifications are restricted to insertions) as long as the grammar has no useless non-terminals.

Similarly, procedure *DynamicSWSF−FP* can be used to solve batch SWSF fixed point problems.

5.3. Maintaining Minimum-Cost Derivations

Procedure *DynamicSSF−G* addresses the problem of maintaining the *cost* of the minimum-cost derivations, and not the problem of maintaining minimum-cost *derivations* themselves. However, the algorithm 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.

5.4. 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 $u_i \to v_i$, for $1 \le i \le k$, 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.

By this method, we obtain a bounded incremental algorithm for the dynamic APSP>0 problem; its running time is $O(\|\delta\| \log \|\delta\|)$.

5.5. 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. Some of the types of changes considered are very special kinds of input changes and might not appear particularly interesting or worth studying, but such changes do appear in the application considered in the next subsection (Section 5.6).

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 [14, 37]. 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 $a \to b$ is replaced by $f(a) + length(a \to b) - f(b)$, then the shortest paths in the graph are unchanged from the original edge-length mapping. If f satisfies the property that $f(a) + length(a \to b) - f(b) \ge 0$ for every edge $a \to b$ 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. For certain problem instances, we can simply define $f(u)$ to be $d_{old}(u)$. First note that the transformation is well-defined only for edges $a \to b$ such that $d_{old}(b)$ is not ∞. For every edge $a \to b$ in the original graph we have $d_{old}(b) \le d_{old}(a) + length_{old}(a \to b)$. Consequently, $d_{old}(a) + length_{old}(a \to b) - d_{old}(b) \ge 0$. Hence, when $d_{old}(b)$ is not ∞, the transformed length of an unmodified edge $a \to b$ will be non-negative. Similarly, if $a \to b$ is a modified edge, the transformed length will be non-negative as long as $length_{new}(a \to b) \ge length_{old}(a \to b)$—*i.e.*, as long as the length of the edge $a \to b$ was not decreased during the input modification—and $d_{old}(b)$ is not ∞.

Thus, by using d_{old} as transformation-function f, our incremental algorithm can be used 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 and no edge-length is decreased. This will work since the length of an edge $a \to b$ 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 one particular 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 these inserted edges are examined only at the beginning of the update, to determine the new value for vertex u; these edges play no role in the subsequent updating of values of other affected vertices. We leave the details to the reader.

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

Yap [38] 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-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 algorithm so obtained is also more general than Yap's algorithm in that it solves the single-source or single-sink problem as well as 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≥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 [16] 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 [17]).

We now consider how our incremental algorithm for the shortest-path problem can be used to solve this problem more efficiently. Let $u_1, \ldots, u_{k'}$ be the set of all vertices in the graph that have an incoming edge of negative length. Thus $k' \le k$. 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 5.5.

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 time bound indicates, since all the update steps have bounded complexity.)

6. 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 4. 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 [24] 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.

Recently, Ausiello *et al* [5]. presented a *semi-dynamic* algorithm for maintaining optimal hyperpaths in directed hypergraphs. This algorithm is quite similar to our algorithm, except that it handles only the insertion of hyperarcs (productions) into the hypergraph (grammar). The relationship between the hypergraph problem and grammar problem is discussed below.

A directed hypergraph consists of a set of nodes and a set of hyperarcs. Each hyperarc connects a set of sources nodes to a single target node. The concept of hyperpaths is defined recursively. There exists an empty hyperpath from a set S of nodes to a node t if $t \in S$. A non-empty hyperpath from a set S of nodes to a node t consists of an hyperarc from a set S' to t and a hyperpath from S to s for every node s in S'. Ausiello *et al.* [5] introduced the concept of a *value-based measure* for hyperpaths. Assume that every hyperarc e has an associated "weight" $wt(e)$. A value-based measure μ is described by a triple (f, ψ, μ_0), where μ_0 is a real value, f is a monotonic, binary, real-valued function, and ψ is a monotonic, commutative, and associative function from sets of reals to reals. The measure $\mu(\emptyset)$ of an empty hyperpath $\emptyset$ is defined to be μ_0; the measure $\mu(P)$ of a non-empty path P that can be recursively decomposed into a hyperedge e and hyperpaths $P_1, \ldots, P_k$ is defined to be $f(wt(e), \psi(\mu(P_1), \ldots, \mu(P_k)))$.

The analogy between context-free grammars and directed hypergraphs should be obvious. Nodes correspond to non-terminals, while hyperarcs correspond to productions. A *hyperpath* in a hypergraph corresponds to a *derivation* in the grammar. The value-based measure of a hyperpath is similar to the cost assigned to a derivation in Knuth's grammar problem, and an optimal hyperpath corresponds to a minimum-cost derivation.

The grammar problem is actually a strict generalization of the (unordered) directed hypergraph problem. In particular, in the grammar problem a richer class of functions are permitted on the hyperedges (and hence as "value-based measure functions"). This comes about because of two ways in which the frameworks differ:

(i) In the grammar problem, different productions θ and τ can have different production functions g_θ and g_τ. In the (unordered) directed hypergraph problem, all hyperedges have functions built from a single function f and a single function ψ.

(ii) In the grammar problem the nonterminals of each production —*i.e.*, the predecessors in each hyperedge—are ordered. Because the predecessors in each hyperedge are unordered, the function ψ, is required to be commutative and associative. This is not necessary in the grammar problem because the order of nonterminals in productions permits making some distinctions among the values that "flow" along the hyperedges. Thus, for example, with the grammar-problem formulation one can use a function such as:

$$g_\theta \triangleq \lambda x, y \,.\, w_\theta + 2x + y.$$

This is not possible in the class of unordered problems that Ausiello et al. deal with.

There is another minor difference between the grammar problem and the hypergraph problem worth mentioning. Derivations of terminal strings from non-terminals really correspond to hyperpaths from an empty set of nodes to a node. Hyperpaths from an arbitrary, non-empty, set of nodes to a node can model "partial derivations", or derivations of sentences containing non-terminals as well. This is, however, not a significant difference. Such hyperpaths can be represented in the grammar by derivations by simply adding epsilon productions for all the relevant non-terminals.

Previous work on algorithms for the dynamic shortest-path problem include papers by Murchland [27, 28], Loubal [26], Rodionov [34], Halder [21], Pape [29], Hsieh *et al.* [22], Cheston [9], Dionne [12], Goto *et al.* [20], Cheston and Corneil [10], Rohnert [35], Even and Gazit [15], Lin and Chang [25], Ausiello *et al.* [2, 4], and Ramalingam and Reps [30]. The work described in this paper differs from these algorithms in several ways. First, the incremental algorithm we have presented is the 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 [20]. The algorithm described in this paper is more efficient and capable of handling more general modifications than the algorithm described in [20]. Finally, we have generalized our algorithm to handle a version of the dynamic fixed point problem. A more comprehensive discussion and comparison of the above-mentioned algorithms appears in [31].

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 WSF grammar problem can be reduced to the problem of computing the maximal fixed point of a collection of WSF equations, and how the SWSF grammar problem can be reduced to the problem of computing the unique fixed point of a collection of SWSF equations. We first show 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.

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

Furthermore, 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,

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

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,

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

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 WSF grammar, then Q_G is a collection of WSF equations, while if G is an SWSF 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 a *w.s.f*. then the above equation is an WSF equation. Similarly, if each g_i is a *s.w.s.f*., then the above equation is a *s.w.s.f*. equation. 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 a WSF grammar with no useless non-terminals then ($m_G(Y)$ | Y is a non-terminal) is a fixed point of Q_G.

Proof.

$$\begin{aligned}
m_G(Y) &= \min_{Y \longrightarrow^* \alpha} val(\alpha) \quad \text{(from the definition of } m_G(Y)\text{, see Section 2)}\\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} \quad \min_{g(X_1, \ldots, X_k) \longrightarrow^* \alpha} val(\alpha)\\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} \min\{val(g(\alpha_1, \ldots, \alpha_k)) \mid X_i \longrightarrow^* \alpha_i\}\\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} \min\{g(val(\alpha_1), \ldots, val(\alpha_k)) \mid X_i \longrightarrow^* \alpha_i\}\\
&\qquad \text{(from the definition of } val(g(\alpha_1, \ldots, \alpha_k)))\\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} g(\min_{X_1 \longrightarrow^* \alpha_1} val(\alpha_1), \ldots, \min_{X_k \longrightarrow^* \alpha_k} val(\alpha_k))\\
&\qquad \text{(since } g \text{ is monotonic, and none of the } X_i \text{ is useless)}\\
&= \min_{Y \longrightarrow g(X_1, \ldots, X_k)} g(m_G(X_1), \ldots, m_G(X_k)) \quad \text{(from the definition of } m_G)
\end{aligned}$$

□

Lemma A.5. Let G be a WSF grammar, and let ($f(Y)$ | Y is a non-terminal) be a fixed point of Q_G. Then, $f(Y) \le m_G(Y)$ 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 $f(Y) \le val(\alpha)$. 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 $f(X_i) \le val(\alpha_i)$. It follows from the monotonicity of g that $g(f(X_1),...,f(X_k)) \le g(val(\alpha_1),...,val(\alpha_k)) = val(\alpha)$. Since ($f(Y)$ | Y is a non-terminal) is a fixed point of Q we have $f(Y) \le g(f(X_1),...,f(X_k))$. The result follows. □

Theorem A.6. Let G be an WSF grammar with no useless non-terminals. 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 3, which computes this fixed point. The uniqueness of the fixed point may be established as follows.

Assume, to the contrary, that (a_i | $1 \le i \le k$) and (b_i | $1 \le i \le k$) 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.

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

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

We now summarize the above results. Theorems A.3 and A.6 establish that the WSF 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.

References

1. Alpern, B., Hoover, R., Rosen, B.K., Sweeney, P.F., and Zadeck, F.K., "Incremental evaluation of computational circuits," pp. 32-42 in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms,* (San Francisco, CA, Jan. 22-24, 1990), Society for Industrial and Applied Mathematics, Philadelphia, PA (1990).
2. Ausiello, G., Italiano, G.F., Spaccamela, A.M., and Nanni, U., "Incremental algorithms for minimal length paths," pp. 12-21 in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms,* (San Francisco, CA, Jan. 22-24, 1990), Society for Industrial and Applied Mathematics, Philadelphia, PA (1990).
3. Ausiello, G., Nanni, U., and Italiano, G.F., "Dynamic maintenance of directed hypergraphs," *Theoretical Computer Science* **72**(3) pp. 97-117 (May 1990).
4. 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).
5. Ausiello, G., Italiano, G.F., and Nanni, U., "Optimal traversal of directed hypergraphs," TR-92-073 (September 1992).
6. Berge, C., *Graphs and Hypergraphs,* North-Holland, Amsterdam (1973).
7. Berge, C., *Hypergraphs: Combinatorics of Finite Sets,* North-Holland, Amsterdam (1989).
8. Carroll, M.D., "Data flow update via dominator and attribute updates," Ph.D. dissertation, Rutgers University, New Brunswick, NJ (May 1988).
9. 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).
10. 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).
11. Dijkstra, E.W., "A note on two problems in connexion with graphs," *Numerische Mathematik* **1** pp. 269-271 (1959).
12. Dionne, R., "Etude et extension d'un algorithme de Murchland," *INFOR* **16**(2) pp. 132-146 (June 1978).
13. 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).
14. Edmonds, J. and Karp, R.M., "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM* **19** pp. 248-264 (1972).
15. Even, S. and Gazit, H., "Updating distances in dynamic graphs," pp. 271-388 in *IX Symposium on Operations Research,* (Osnabrueck, W. Ger., Aug. 27-29, 1984), *Methods of Operations Research,* Vol. 49, ed. P. Brucker and R. Pauly,Verlag Anton Hain (1985).
16. 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).
17. 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).
18. Gallo, G., Longo, G., Pallottino, S., and Nguyen, S., "Directed hypergraphs and applications," *Discrete Applied Mathematics* **42** pp. 177-201 (1993).
19. Gondran, M. and Minoux, M., *Graphs and Algorithms,* John Wiley and Sons, New York (1984).
20. Goto, S. and Sangiovanni-Vincentelli, A., "A new shortest path updating algorithm," *Networks* **8**(4) pp. 341-372 (1978).
21. Halder, A.K., "The method of competing links," *Transportation Science* **4** pp. 36-51 (1970).
22. 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).

23. Italiano, G.F. and Nanni, U., "On line maintenance of minimal directed hypergraphs," pp. 335-349 in *Proceedings 3rd Convegno Italiano di Informatica Teorica (Mantova)*, World Science Press (1989).

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

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

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

27. 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).

28. 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 ().

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

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

31. Ramalingam, G. and Reps, T., "An incremental algorithm for a generalization of the shortest-path problem," TR-1087, Computer Sciences Department, University of Wisconsin, Madison, WI (May 1992).

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

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

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

35. 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).

36. 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).

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

38. 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).