Constraints, 10, 159–183, 2005

Maintaining Longest Paths Incrementally

IRIT KATRIEL irit@mpi-sb.mpg.de
Max-Plank-Institut für Informatik, Saarbrücken, Germany

LAURENT MICHEL ldm@engr.uconn.edu
University of Connecticut, Storrs, CT 06269-2155, USA

PASCAL VAN HENTENRYCK pvh@cs.brown.edu
Brown University, Box 1910, Providence, RI 02912, USA

Abstract. Modeling and programming tools for neighborhood search often support invariants, i.e., data structures specified declaratively and automatically maintained incrementally under changes. This paper considers invariants for longest paths in directed acyclic graphs, a fundamental abstraction for many applications. It presents bounded incremental algorithms for arc insertion and deletion which run in $O(\|\delta\| + |\delta|\log|\delta|)$ time and $O(\|\delta\|)$ time respectively, where $|\delta|$ and $\|\delta\|$ are measures of the change in the input and output. The paper also shows how to generalize the algorithm to various classes of multiple insertions/deletions encountered in scheduling applications. Preliminary experimental results show that the algorithms behave well in practice.

Keywords: incremental, longest path, heaviest path, bounded computation, graph, constraint, local search, scheduling

1. Introduction

The last decades have seen significant progress in the design and implementation of modeling and programming tools for combinatorial optimization. Historically, the major focus of that research has been on systematic search (e.g., constraint satisfaction and mathematical programming), but recent years have seen increased attention being devoted to local search and its variations (See, for instance, [7, 11, 13, 22, 25, 27]).

The design of modeling and programming tools for local search generally involves abstractions to express the neighborhood and to encapsulate incremental algorithms. Localizer [13] proposed the concept of invariants, which specifies, in a declarative fashion, data structures that are then maintained incrementally by the system. Invariants were used subsequently in [11, 26]. More recently, constraint-based approaches to local search (e.g., [3, 8, 14, 27]) were proposed, where constraints incrementally maintain properties such as their violation degrees. The Comet system [12] pushed this idea further and introduced the concept of differential objects, which can be viewed as the counterpart of global constraints for local search. Differentiable objects not only maintain properties incrementally, but also make it possible to evaluate the effects of various actions (or moves) on these properties (e.g., swapping the values of two variables), since such queries are often used to choose appropriate moves in local search algorithms. In general, differentiable objects capture combinatorial substructures of the application at hand and they were instrumental in finding novel, more efficient, algorithms for several combinatorial optimization problems [12, 16].

This paper was motivated by differentiable objects for scheduling applications, where it is often critical to maintain longest paths in directed acyclic graphs (DAGs) in order to evaluate the makespan or, more generally, earliest and latest completion times. These longest paths are then used in list or bidirectional scheduling (e.g., [5]), in insertion heuristics (e.g., [28]), as well as in neighborhood search (e.g., [1, 5, 17]). For instance, a key component of many of these algorithms is the ability to update the makespan after an insertion or to evaluate the impact of swapping two tasks on the makespan. Note also that, in some scheduling applications, arcs with negative weights may be induced by specific distance constraints.

The main technical result of this paper are novel algorithms to maintain longest paths in directed acyclic graphs under arc insertions and deletions. The paper presents bounded incremental algorithms for these two operations which run in time $O(\|\delta\| + |\delta| \log|\delta|)$ (insertion) and $O(\|\delta\|)$ (deletion), where $\|\delta\|$ represents the *size of the changes in the input and output*. The results use the Bounded Incremental Computation (BIC) model of Ramalingam and Reps [19]. The BIC model differentiates more incremental algorithms than the traditional online computation model, which only analyzes algorithms in terms of the input size. The BIC model is particularly appropriate for heuristic and neighborhood search, where the change in the output is often small compared to the total input size. The paper also shows how to adapt these algorithms for important operations in scheduling and gives preliminary experimental results indicating the practicality of the algorithms.

The rest of the paper is structured as follows. Section 2 gives an overview of the BIC model. Section 3 discusses the intuition behind the algorithms. Sections 4 and 5 describe the algorithms in the case of strictly positive arc weights and give their correctness proofs. Section 6 generalizes the algorithm for some applications in scheduling. Section 7 presents the generalization that handles arcs of any weight. Section 8 gives some preliminary experimental results, Section 9 describes related work, while Section 10 concludes the paper. In the rest of the paper, when the meaning is clear from the context, we will sometimes abuse language and talk about "longest path" to mean either the path or the length of the path.

2. Bounded Incremental Computation

At a high level of abstraction, incremental algorithms can be modelled as updating the output of a function subject to changes to its input. Let f be a function, x be an input, and ϵ be a change in x. An incremental algorithm receives x, $f(x)$, and ϵ as inputs and transforms $f(x)$ into $f(x+\epsilon)$, where $x+\epsilon$ denotes the result of applying change ϵ on input x. For instance, x may be a directed graph with a source, f may be a function which computes the length of the longest path from the source to each of the vertices, and ϵ

may be the insertion of an arc $a \rightarrow b$ or the removal of such an arc. In general, it is useful in incremental algorithms to maintain auxiliary information in order to compute $f(x + \epsilon)$. Provided that the auxiliary information is polynomially related in size to the output, the problem can then be viewed as computing an enhanced function f' incrementally. As a consequence, we can safely ignore this issue without loss of generality and work directly with f'.

Various models for analyzing incremental algorithms have been proposed and they include online algorithms, amortized analysis [23], and *bounded incremental computation* (BIC) [19]. Many such models analyze the complexity of incremental algorithms in terms of the input size (e.g., $x + \epsilon$). *The BIC model, on the contrary, studies the behavior of incremental algorithms in terms of the changes in the input and output.* As a consequence, the BIC model has a finer granularity and can differentiate algorithms that other models cannot. In addition, it is particularly appropriate in the context of neighborhood search, where most of the neighborhood generally remains unchanged from one iteration to the next. Analyzing incremental algorithms in terms of the neighborhood size is thus not very informative in general.

Since this paper assumes the BIC model, let us describe its main concepts more precisely. Let $\Delta(f, x, \epsilon)$ denote the change between $f(x)$ and $f(x + \epsilon)$ and let $\delta(f, x, \epsilon)$ denote $\epsilon + \Delta(f, x, \epsilon)$. For instance, in an incremental longest path algorithm, $\Delta(f, x, \epsilon)$ may represent the pairs (vertex, length) which have changed when ϵ (e.g., an arc insertion) is performed. Since, in general, the function f and the change ϵ are clear from the context, we use Δ and δ for simplicity. The BIC model analyzes the performance of an algorithm in terms of $\|\delta\|$, i.e., a measure of the size of δ. The measure $\|\delta\|$ may actually be greater than $|\delta|$ for reasons that will become clear shortly, but it is, in general, closely related.

An incremental algorithm is *bounded* if, for all input x and any allowed change ϵ, its execution time depends only on δ, not the size of the entire input $x + \epsilon$. It is *unbounded* otherwise. Many incremental problems are unbounded (e.g., graph reachability under the local persistent model [19]) and hence the existence of a bounded algorithm is a strong guarantee for incremental performance.

An example of a bounded incremental algorithm is the shortest path algorithm of Ramalingam and Reps [19], which runs in $O(\|\delta\| \log \|\delta\|)$ time for arc insertions and deletions, when the arc weights are strictly positive. The complexity is defined in terms of the *affected vertices*, i.e., the vertices whose shortest paths have changed, and their adjacent arcs. More precisely, $\|\delta\|$ denotes the number of affected vertices plus the number of their adjacent arcs. The complexity is not expressed in terms of the number of affected vertices only, denoted by $|\delta|$, since it is reasonable to assume that any algorithm would necessarily have to examine the vertices that are adjacent to an affected vertex, in order to determine if they are affected as well. For graphs with bounded degrees (e.g., jobshop scheduling), this issue is of course moot.

This paper presents a bounded algorithm that maintains the longest paths in a DAG. The algorithm takes $O(|\delta| \log |\delta| + \|\delta\|)$ time for an arc insertion and $O(\|\delta\|)$ time for an arc deletion. The paper also discusses several generalizations of this result, including the insertion/deletion of multiple arcs and the detection of cycles.

3. Intuition

In this section we give the high-level intuition behind the first algorithms presented in this paper and explain why some simple and natural ideas do not lead to bounded algorithms. We initially focus on graphs with strictly positive arc weights. This restriction is lifted in Section 7. Throughout the paper, we use directed acyclic graphs with a source s.[1] Given a DAG $G = (V, A)$ and a vertex $v \in V$, we denote by $lp\,(G, v)$ the length of a longest path from the source of G to vertex v. The projection of a graph $G = (V, A)$ wrt its longest paths is the graph $G_{|l} = (V, A')$ where

$$A' = \{x \to y \mid lp(G,x) + d(x,y) = lp(G,y)\},$$

i.e., the subgraph consisting of all arcs belonging to longest paths.

For a DAG $G = (V, A)$, let $pred\,(G, v) = \{u \mid (u, v) \in A\}$, $succ\,(G, v) = \{u \mid (v, u) \in A\}$ and for every $u, v \in A$, let $d\,(u, v)$ be the weight of the arc (u, v). Figure 1 presents an offline algorithm that computes, for each node $v \in V$, the weight $l\,(v)$ of the longest path from the source of G to v. The total running time of the algorithm is $O\,(|V| + |A|)$ and its key idea is to consider the vertices in topological order, which guarantees that, when a vertex is considered, its predecessors have the correct longest path values. Lines 1–2 compute the initial indegree of the vertices and Line 3 inserts the source into the queue. Lines 4 and 5 dequeue a vertex and compute the length of its longest path from the source. Lines 7 to 9 decrement the indegrees of the successors of v and insert each of them into the queue if all of its predecessors have been updated, i.e., when its indegree is 0.

Consider now the problem of updating the longest paths after the insertion of an arc $a \to b$. To obtain a bounded algorithm, it is necessary to consider affected vertices only, i.e., those vertices whose longest paths have changed. Figure 2 depicts such a situation. The affected vertices are shown in the grey area. Note that vertex g is not affected, although one of its predecessors is. The reason is that the new longest path through f is not longer than the longest path through e.

```
1.  forall($v \in V$) do
2.     $indegree(v) = |pred(G,v)|$;
3.  $Q = \{v \mid indegree(v) = 0\}$;
4.  while $Q \neq \emptyset$ do
5.     $v = dequeue(Q)$;
6.     $l(v) = max(w \in pred(G,v))\ l(w) + d(w,v)$;
7.     forall $w \in succ(G,v)$ do
8.        $indegree(w) = indegree(w) - 1$;
9.        if $indegree(w) = 0$ then
10.          $insert(Q,w)$;
```

Figure 1. An offline algorithm for longest path in a DAG.

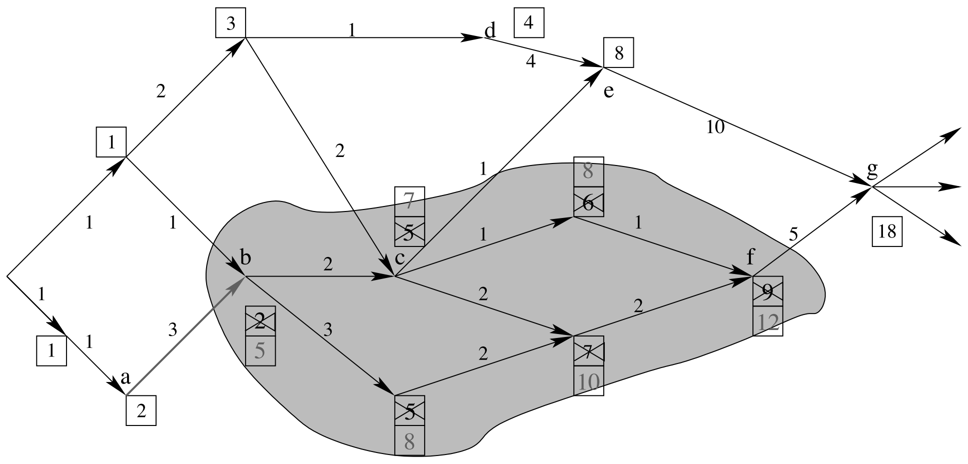

Figure 2. The affected set of an insertion.

Since the offline algorithm works in terms of degrees, it would be ideal to apply the offline algorithm on the subgraph consisting of the affected vertices. Unfortunately, as vertex g indicates, identifying the set of affected vertices requires the computation of longest paths.

Another natural approach would be to maintain a topological ordering incrementally and to use this topological ordering to propagate the changes to the longest paths. The use of indegrees in the offline algorithm is, in fact, a simple way to order the vertices topologically. This approach is appealing, since there exists a bounded incremental algorithm for priority ordering which can be used for that purpose [2, 10]. Unfortunately, this simple idea does not lead to a bounded algorithm. Indeed, a change to the topological ordering does not necessarily entail a change to the longest paths, so that the incremental algorithm for topological ordering may consider non-affected vertices. For instance, if successive integers are used as topological numbers, the arc insertion $a \longrightarrow b$ would change the topological number of g and its successors, although they are not affected vertices for the longest paths. Similar examples can of course be produced for other choices of topological numbers.

The key idea behind our insertion algorithm is the observation that the lengths of the longest paths in the graph G^- before the insertion are, in fact, a topological order for the affected vertices, since the longest path of a vertex is necessarily greater than the longest paths of its predecessors. As a consequence, it is possible to adapt the offline algorithm in order to propagate the changes to the longest paths using that topological ordering, i.e., to enqueue the successors of affected vertices when the lengths of their longest paths are increasing and to dequeue them by non-decreasing order of their previous longest path lengths. Such an algorithm is shown in Figure 3. Let G^- be the graph G at call time. Line 2 tests whether the new arc $x \longrightarrow y$ changes the longest path of its destination y. If it does, then y is inserted into the queue with $l\,(y)$, its longest path in

```
procedure insertArc($G$,$x \to y$)
begin
1.  $G = G \cup \{x \to y\}$;
2.  if $l(x) + d(x,y) > l(y)$ then
3.     $insert(Q, \langle l(y), y\rangle)$;
4.     while $Q \neq \emptyset$ do
5.        $v = extractMin(Q)$;
6.        $l(v) = max(x \in pred(G,v))\ l(x) + d(x,v)$;
7.        forall($w \in succ(G,v)$) do
8          if $l(v) + d(v,w) > l(w)$ then
9.             if $w \notin Q$ then $insert(Q, \langle l(w), w\rangle)$;
end
```

Figure 3. A Preliminary version of procedure `insertArc`.

G^{-}, as its key. The affected vertices are computed and processed in lines 5–9. Line 5 extracts the vertex *v* with the smallest key and updates its longest path. It then considers each successor *w* of *v* and inserts *w* in the queue if its longest path increases and it is not in the queue already. The algorithm runs in time $O(|\delta|\log|\delta| + \|\delta\|)$ using a priority queue. It only uses *insert* and *extractMin* on the queue (not *updateKey*, which changes the key of an item in the queue) and each affected vertex enters the queue once.

The key idea behind deletion is rather different. The algorithm relies on the fact that the affected vertices can be identified without computing longest paths. This is possible because the length of a longest path decreases: it is not necessary to know by how much. More precisely, arc deletion can be thought of as working on subgraphs $G_{|l}$ obtained by

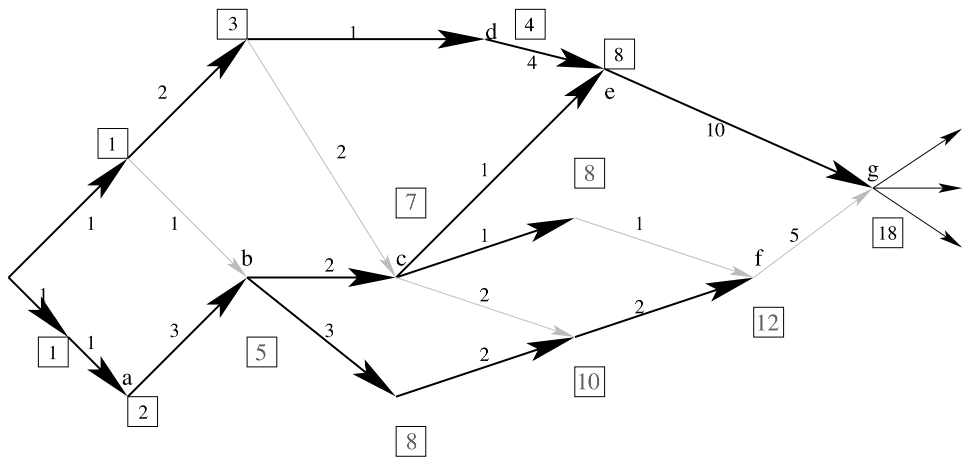

Figure 4. The longest path projection $G_{|l}$.

keeping only those arcs that belong to longest paths. If a vertex *v* is affected and *w* is one of its successors in $G_{|l}$, vertex *w* is affected if $v \to w$ is the only arc incoming into *w* in $G_{|l}$. By proceeding this way, all affected vertices can be computed in $O(\|\delta\|)$ time. Figure 4 depicts the graph $G_{|l}$ from our previous example. Consider the deletion of $a \to b$ which obviously affects *b*. Its successor *c* is also affected, since it has only one incoming arc in $G_{|l}$. On the other hand, vertex *e* is not affected since it has an incoming arc whose origin is not affected. Once the affected vertices are computed, arc deletion can proceed simply by applying the offline algorithm on the affected vertices. Of course, the above discussion indicates that $G_{|l}$ (or at least the indegrees in $G_{|l}$) must be maintained incrementally. As we will see, maintaining $G|_l$ does not increase the complexity of the algorithms. The rest of the paper presents these algorithms in detail, together with the correctness proofs and some important generalizations. Once again, we focus on strictly positive weights, this restriction being lifted in Section 7.

4. Insertion

Figure 5 depicts procedure `insertArc`. The main differences compared to the preliminary version presented earlier are lines 7–8 and 12–15, which maintain the projected graph. Lines 7–8 update the projected graph for an affected vertex *v*, lines 12–13 add an arc originating from an affected vertex to a non-affected vertex, while lines

```
procedure insertArc($G$,$x \to y$)
begin
1.   $G = G \cup \{x \to y\}$;
2.   if $l(x) + d(x,y) > l(y)$ then
3.     $insert(Q, \langle l(y), y\rangle)$;
4.     while $Q \neq \emptyset$ do
5.       $v = extractMin(Q)$;
6.       $l(v) = max(x \in pred(G,v))\ l(x) + d(x,v)$;
7.       $G_l = G_l \setminus \{x \to v \mid x \to v \in G_l\}$;
8.       $G_l = G_l \cup \{x \to v \mid x \in pred(G,v) \wedge l(x) + d(x,v) = l(v)\}$;
9.       forall($w \in succ(G,v)$) do
10.        if $l(v) + d(v,w) > l(w)$ then
11.          if $w \notin Q$ then $insert(Q, \langle l(w), w\rangle)$;
12.        else if $l(v) + d(v,w) = l(w)$ then
13.          $G_l = G_l \cup \{v \to w\}$;
14.  else if $l(x) + d(x,y) = l(y)$ then
15.    $G_l = G_l \cup \{x \to y\}$;
end
```

Figure 5. Procedure `insertArc`.

14–15 handle the case of the inserted arc. We now prove the correctness of the algorithm. We first define formally the set of vertices affected by an arc insertion.

Definition 1 (Affected Vertices) Let $G = (V, A)$, $x \rightarrow y \notin A$, and $G' = (V, A \cup \{x \rightarrow y\})$. The set of vertices affected by the insertion of $x \rightarrow y$ into G is defined as

$$Aff_I(G, x \rightarrow y) = \{v \in V | lp(G', v) > lp(G, v)\}.$$

In the following, we abuse notation and omit the arguments of Aff_I when they are clear from the context. The following proposition states that a vertex is affected only if one of its predecessors is affected or it is the target of the inserted arc.

Proposition 1 *Let* $G = (V, A)$, $x \rightarrow y \notin A$, *and* $G' = (V, A \cup \{x \rightarrow y\})$. *Then,*

$$w \in Aff_I(G, x \rightarrow y) \Rightarrow \exists v \in pred(G', w) : lp(G', v) + d(v, w) > lp(G, w).$$

Proof 1: By definition of longest paths, we have that

$$lp(G', w) = \max(v \in pred(G', w)) lp(G', v) + d(v, w).$$

Assume that

$$\forall v \in pred(G', w) : lp(G', v) + d(v, w) \leq lp(G, w).$$

It follows that $lp\,(G', w) \leq lp\,(G, w)$ which contradicts

$$w \in Aff_I(G, x \rightarrow y).$$

The proposition makes it natural to define a binary relation aff_I. ■

Definition 2 Let G = (V, A), $x \rightarrow y \notin A$, and $G' = (V, A \cup \{x \rightarrow y\})$. The binary relation aff_I is defined as

$$aff_I(v, w) \Leftrightarrow lp(G', v) + d(v, w) > lp(G, w) \wedge v \in pred(G', w).$$

We use $aff_I{*}$ to denote the transitive closure of aff_I.

The following proposition concerns the affected vertices.

Proposition 2 *Let* $G = (V, A)$, $x \rightarrow y \notin A$, $G' = (V, A \cup \{x \rightarrow y\})$, *and let* $v \in aff_I\,(G, x \rightarrow y)$ $(v \neq y)$. *Then,* $aff_I{*}(y, v)$ *holds, i.e., there exists a path of affected vertices from* y *to* v.

Proof 2: The proof is by induction on the maximum number of arcs d on a path from y to affected vertices. The result holds when $d = 1$, i.e., the successors of y that are affected. Assume that the result holds for $d \leq k$ and consider an affected vertex w at distance $d = k + 1$. By Proposition 1, vertex w has an affected predecessor $v \in pred\,(G', w)$ such that

$$lp(G', v) + d(v, w) > lp(G, w)$$

and hence $aff_I(v, w)$ holds. By induction, $aff_I{}^*(y, v)$ holds and the result follows. ∎

Definition 3 (Specification of `insertArc`*)* Let G = (V, A) be a DAG with strictly positive arc weights, $x \to y \notin A,$ and $G' = (V, A \cup \{x \to y\})$. Procedure `insertArc` $(G, x \to y)$ satisfies the following specification:

$$Pre : \forall v \in V : l(v) = lp(G, v) \wedge G_l = G_{|l}.$$
$$Post : \forall v \in V : l(v) = lp(G', v) \wedge G_l = G'_{|l}.$$

Theorem 1 *Procedure* `insertArc` *is correct and terminates.*

Proof 3: The proof relies on the observation that the algorithm partitions the affected vertices in three sets

$$P = \{x \in Aff_I \mid l(x) = lp(G', x)\}; \tag{1}$$
$$Q = \{x \in Aff_I \mid \exists v \in P : v \to x \;\&\; x \notin P\}; \tag{2}$$
$$R = \{x \in Aff_I \mid \exists v \in Q : aff_I{}^*(v, x) \;\&\; x \notin P \cup Q\} \tag{3}$$

and that the following two invariants hold at line 4 in the algorithm

$$Aff_I = P \cup Q \cup R \qquad \forall v \in P, \forall x \in Q : lp(G, v) \leq lp(G, x).$$

Initially, $P = \emptyset$, $Q = \{y\}$, $R = Aff_I \setminus \{y\}$, and the invariants hold by Proposition 2. Assume now that the invariants hold at iteration i. We show that lines 5–13 restore the invariants for iteration $i + 1$. Line 5 extracts the vertex v with the smallest value $l\,(v) = lp\,(G, v)$ from Q. Since $lp\,(G, v) > lp\,(G, p)$ for all $p \in pred\,(G, v)$, all its affected predecessors must be in P by Invariant (2) and the fact that

$$\forall s \in succ(G, v) : lp(G, v) < lp(G, s).$$

As a consequence, line 6 correctly computes $l\,(v) = lp\,(G', v)$. Each successor w of v now belongs to $Q \cup R$ by Invariant (2) and lines 8–10 move these successors of v from R to Q, since $v \in P$ after line 6. Observe that no new vertices are added to the union $Q \cup R$ and hence Invariant (1) is preserved. By selection of v and since $\forall y \in succ$

(G, x): $lp(G, x) < lp(G, y)$, Invariant (2) holds as well. On termination, Q is empty, which entails that R is empty, and hence $l(v) = lp(G', v)$ for all $v \in V$. The algorithm is also guaranteed to terminate, since the size of $Q \cup R$ strictly decreases at each iteration. It is easy to verify that G_l is also updated correctly, since it is recomputed for each affected vertex (lines 7–8) and since arcs to successors of affected vertices are inserted in lines 13 and 15. ■

By the proof of Theorem 1, an affected vertex is inserted into, and extracted from, the queue at most once. No other operation (e.g., *updateKey*, which changes the key of an item in the queue) is used by the algorithm. As a consequence, the algorithm runs in time $O(|\delta| \log |\delta| + \|\delta\|)$ using a priority queue, since it must consider all adjacent vertices of an affected vertex to determine whether they are themselves affected.

5. Arc Deletion

Figures 6 and 7 depict the algorithms to handle the deletion of an arc $x \rightarrow y$. Function `computeAffected` in Figure 6 identifies the set of affected vertices. It starts with the deleted arc $x \rightarrow y$ and works on the projected graph. Each iteration dequeues an affected vertex and inserts its successors in the queue if they are affected: A successor w is affected if all of its predecessors in the projected graph are affected. This is tested by removing from G_l all arcs $v \rightarrow w$, where v is affected. When a vertex has no predecessor in G_l, it is affected. Procedure `removeArc` in Figure 7 is the main routine. If the deletion of $x \rightarrow y$ affects y, the procedure computes the affected vertices using Function `computeAffected`. It then initializes the indegrees of all affected vertices using the affected vertices only. Indeed, the unaffected vertices can be considered as having been processed, since the lengths of their longest paths did not change. It then applies the

```
function computeAffected($G_l, y$)
begin
1.  $Q = \{y\}$;
2.  $A = \emptyset$;
3.  while $Q \neq \emptyset$ do
4.     $u = dequeue(Q)$;
5.     $A = A \cup \{u\}$;
6.     forall($v \in succ(G_l, u)$) do
7.        $G_l = G_l \setminus \{u \rightarrow v\}$;
8.        if $pred(G_l, v) = \emptyset$ then
9.           $insert(Q, v)$;
10. return $A$;
end
```

Figure 6. Function `computeAffected`.

```
procedure removeArc($G$,$x \to y$)
begin
1.  $G = G \setminus \{x \to y\}$;
2.  if $x \to y \in G_l$ then
3.     $G_l = G_l \setminus \{x \to y\}$;
4.     if $pred(G_l, y) = \emptyset$ then
5,        $Affected$ = computeAffected($G_l, y$);
6.        forall($v \in Affected$) do
7.           $indegreelp(v) = |pred(G, v) \cap Affected|$;
8.        $Q = \{v \in Affected \mid indegreelp(v) = 0\}$;
9.        while $Q \neq \emptyset$ do
10.          $v = dequeue(Q)$;
11.          $l(v) = max(x \in pred(G, v))\ l(x) + d(x, v)$;
12.          $G_l = G_l \cup \{x \to v \mid x \in pred(G, v) \wedge l(x) + d(x, v) = l(v)\}$;
13.          forall($w \in succ(G, v) \cap Affected$) do
14.             $indegreelp(w) = indegreelp(w) - 1$;
15.             if $indegreelp(w) = 0$ then $insert(Q, w)$;
end
```

Figure 7. Procedure `removeArc`.

offline algorithm on the affected vertices. We now formalize the various concepts and give the correctness proofs.

Definition 4 (Affected Vertices) Let $G = (V, A)$, $x \to y \in A$, and $G' = (V, A \setminus \{x \to y\})$. The set of vertices affected by the deletion of $x \to y$ from G is defined as

$$Aff_D(G, x \to y) = \{v \in V | lp(G', v) < lp(G, v)\}.$$

As before, we abuse notation and omit the arguments of Aff_D when they are clear from the context. We also denote by $x \to_l y$ an arc in $G_{|l}$ and by $x \to_l^* y$ the existence of a path from x to y in $G_{|l}$. The following proposition is the counterpart to Proposition 1 and states that a vertex is affected if and only if all of its predecessors in the projected graph are affected.

Proposition 3 *Let $G = (V, A)$, $x \to y \in A$, $G' = (V, A \setminus \{ x \to y \})$, and let $v \in V$ such that $v \neq y$. Vertex v is affected iff*

$$\forall p \in pred(G_{|l}) : p \in Aff_D(G, x \to y).$$

Proof 4: By definition, v is affected iff $lp(G', v) < lp(G, v)$ which is equivalent to $\forall p \in pred(G, v)$: $lp(G', p) + d(p, v) < lp(G, v)$. Since

$$\forall p \in pred(G, v) \setminus pred(G_{|l}, v) : lp(G, p) + d(p, v) < lp(G, v)$$

and since $lp(G', p) \leq lp(G, p)$, it follows that v is affected iff $\forall p \in pred(G_{|l}, v)$: $lp(G', p) + d(p, v) < lp(G, v)$ which is equivalent to $\forall p \in pred(G_{|l}, v)$: $lp(G\prime, p) < lp(G, p)$. The result follows. ■

Corollary 1 *Let $G = (V, A)$, $x \to y \in A$, $G' = (V, A \backslash \{x \to y\})$, and let $v \in V$ such that $v \neq y$. Vertex v is affected implies $y \to_l^* v$.*

Proof 5: Suppose that no such path exists. Then a longest path to v cannot go through y. By Proposition 3, the source must be affected, which is impossible. ■

Definition 5 (Specification of `computeAffected`) Let $G = (V, A)$ be an arbitrary DAG, $x \to y \in A$, $G' = (V, A \backslash \{x \to y\})$, and $lp(G', y) < lp(G, y)$. Procedure `computeAffected` $(G, x \to y)$ satisfies the specification:

$$Pre : G_l = G_{|l}.$$

$$Post : G_l = G'_{|l} \backslash \{v \to w | v \in Aff_D\}; \text{ the function returns } Aff_D.$$

Theorem 2 *Procedure `computeAffected` is correct and terminates.*

Proof 6: The proof relies on the observation that the algorithm partitions the affected vertices in three sets P, Q, and R, satisfying the invariants

$$v \in P \Rightarrow v \in Aff_D \tag{4}$$

$$v \in Q \Rightarrow v \in Aff_D \tag{5}$$

$$R = \{w \in Aff_D \backslash (P \cup Q) | \exists v \in Q : v \to_l^* v\} \tag{6}$$

$$G_l = G_l \backslash \{v \to w | v \in P \cup Q\} \tag{7}$$

in line 3 of the algorithm. Initially, P is empty, $Q = \{y\}$, and the invariants hold by Corollary 1. By Invariant (5), lines 4 and 5 are correct. Moreover, if v is a successor of u and the test on line 8 succeeds, by Invariant (7), all predecessors of v must be in $P \cup Q$ and are affected. By Proposition 3, v is affected and line 9 is correct. Moreover, all other affected vertices are still reachable from vertices in Q. Indeed, if the only path to an affected vertex w not in $P \cup Q$ goes through u, i.e., $y \to_l \ldots \to_l u \to_l s \to_l \ldots \to_l w$, then s is in Q (because of lines 8–9) and $s \to_l^* w$. On termination, Q is empty and P is the set of affected vertices. The algorithm terminates, since $|Q \cup R|$ strictly decreases at each iteration. ■

Definition 6 (Specification of `removeArc`*)* Let $G = (V, A)$ be an arbitrary DAG, $x \rightarrow y \in A$, and $G' = (V, A \setminus \{x \rightarrow y\})$. Procedure `removeArc` $(G, x \rightarrow y)$ satisfies the specification:

$$Pre : \forall v \in V : l(v) = lp(G, v) \wedge G_l = G_{|l}.$$

$$Post : \forall v \in V : l(v) = lp(G', v) \wedge G_l = G'_{|l}.$$

Theorem 3 *Procedure* `removeArc` *is correct and terminates.*

Proof 7: The proof follows from Theorem 2, the fact that the indegrees for the non-affected vertices are initialized correctly and the correctness of the offline algorithm. ■

Note that the complexity of arc deletion is $O(\|\delta\|)$, since there is no need of a priority queue. Only vertices that are ready to be processed are pushed on the queue in functions `computeAffected` and `removeArc`.

6. Applications to Scheduling

This section describes some generalizations of the algorithms that are important in a variety of scheduling application.

6.1. Multiple Insertions/Deletions

It is easy to generalize the insertion algorithm to accommodate a set of arcs of the form $\{x \rightarrow y_1, \ldots, x \rightarrow y_n\}$. Indeed, since all these arcs have the same origin, the values $lp(G, v)$ are still a valid topological ordering for the affected vertices, since no new topological constraints are introduced between the affected vertices. Such multiple insertions are typical in list-scheduling and bidirectional search algorithms for jobshop scheduling [5]. This suggests that, as long as insertions/deletions do not change the topological ordering, adaptations of Procedure `insertArc` may be used.

Consider for instance changing (increasing or decreasing) the weights of a set of arcs of the form $\{x \rightarrow y_1, \ldots, x \rightarrow y_n\}$, i.e., changing $d(x, y_1), \ldots, d(x, y_n)$. Obviously, the lengths of longest paths $lp(G, v)$ provide a topological ordering of the graph, since the graph has not changed (only the weights). We can thus apply an algorithm similar to `insertArc` in order to propagate the changes to vertices in $\{y_1, \ldots, y_n\}$. The core of such an algorithm is depicted in Figure 8 and is essentially similar to `insertArc`. The main difference is in line 6, which tests whether the lengths have changed (i.e., have been increased or decreased). This procedure may be called with S initialized to those vertices in $\{y_1, \ldots, y_n\}$ which are affected.

A more complex use of multiple insertions/deletions arises in local search algorithms for jobshop or openshop scheduling. Here a typical move consists of swapping two vertices (or tasks) on a critical path which are executing on the same machine. Observe

```
procedure propagateChanges($G$,$S$)
begin
1.    $Q = S$;
2.    while $Q \neq \emptyset$ do
3.       $v = extractMin(Q)$;
4.       $l(v) = max(x \in pred(G,v))\ l(x) + d(x,v)$;
5.       forall($w \in succ(G,v)$) do
6           if $l(v) + d(v,w) \neq l(w)$ then
7.             if $w \notin Q$ then $insert(Q, \langle l(w), w\rangle)$;
end
```

Figure 8. Procedure `propagateChanges`.

that swapping two such vertices is guaranteed not to create cycles [1] and that evaluating the impact of such moves on the makespan for a restricted set of vertices is the basic operation of the successful tabu-search algorithm of Nowicki [17]. The left side of Figure 9 depicts such a situation. In the figure, p_v, v, w, s_w are executed on the machine, and $sj(v)$ and $sj(w)$ represent the job successors of vertices v and w. Such a move seems rather complex. However, observe that we can add an arc $p_v \rightarrow w$ with weight $d(p_v, v) + d(v, w)$ in constant time, since no vertex is affected. We can now remove $v \rightarrow w$ in constant time since, again, no vertex is affected. Now the effect of swapping v and w on the makespan is achieved simply by modifying the weights of $p_v \rightarrow v$ and $p_v \rightarrow w$ appropriately. *As a consequence, algorithm* `propagateChanges` *gives us a bounded* $O(|\delta|\log|\delta| + \|\delta\|)$ *incremental algorithm for evaluating changes to the makespan when swapping two critical vertices*. Of course, none of the above arc operations need to take place in practice. It is sufficient to apply `propagateChanges` on the affected vertices. Similar reasoning can be applied to many more situations, including moves in the neighborhood NB in [5] and arc additions in insertion algorithms [28] for scheduling. Observe also that our deletion algorithm supports multiple deletion naturally, since it only reasons on the projected graph.

6.2. *Cycle Detection*

It is also easy to generalize our algorithm to detect cycles. Since procedure `insertArc` guarantees that a vertex can only be processed once, it suffices to mark the vertices

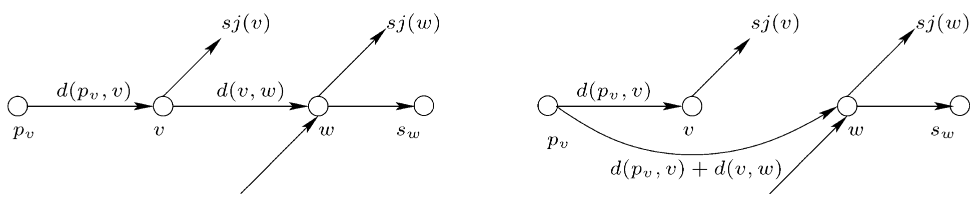

Figure 9. Inverting two vertices on a critical path.

extracted from the queue. A cycle is detected if such a vertex is about to be reinserted into the queue.

7. Generalization to Zero and Negative Weights

First, it is important to notice that the correctness of `computeAffected` and `removeArc` do not depend on the absence of zero or negative arc weights. Indeed, function `computedAffected` depends only on the topology of G_l, while `removeArc` considers a projection of G on the set of affected vertices and proceeds through the vertices of this projection in topological order. As a consequence, it is only necessary to generalize `insertArc` to arbitrary arc weights. This section presents such an algorithm, starting with the intuition before formalizing the algorithm and its correctness.

7.1. Intuition

Recall that the critical issue is to develop an algorithm where every affected vertex is considered once. Because the weights may be non-positive, the lengths of the longest paths in the graph do not constitute a topological order. It is thus necessary to find another ordering of the vertices which considers the affected vertices exactly once and guarantees correctness.

The key observation behind the algorithm is the fact that the variations in the longest paths of the vertices are monotonically non-increasing along paths in the graph. More precisely, if $\langle v_1, \ldots, v_k \rangle$ is a longest path in G', the variations

$$\Delta(v_i) = lp(G', v_i) - lp(G, v_i)$$

satisfy

$$\Delta(v_1) \geq \Delta(v_2) \geq \ldots \geq \Delta(v_k).$$

Figure 10 shows the impact of adding the dashed arc to the DAG. The affected nodes are inside the ellipse and the changes to $lp(v)$ values are shown. Observe that the changes decrease monotonically along paths in the graph. Indeed, the variation to the longest path of a vertex is never larger than the maximum variation of its predecessors.

As a consequence, the algorithm considers the affected vertices in non-increasing order of their variations. The vertex with the largest variation, even if it is not the smallest topologically, cannot increase further along another path, since the variations in such a path are monotonically non-increasing. Of course, the algorithm does not “know” the variations initially but discovers them, together with the affected vertices, as it proceeds. The structure of the algorithm remains essentially the same. Affected vertices are discovered as the algorithm proceeds and vertices are inserted into a priority queue. The priority queue must be implemented using a Fibonacci heap [6] (or another priority

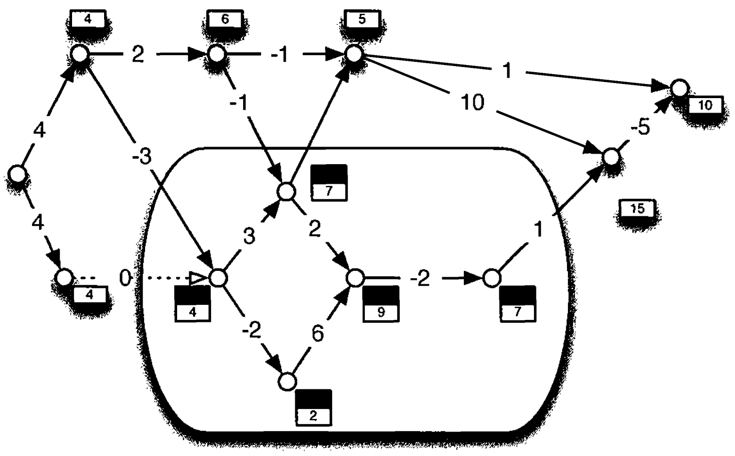

Figure 10. The impact of inserting the dashed arc to the DAG.

queue that enables to update a key in constant time), since the variation of a node may be updated several times after it is inserted into the queue. In addition, it is not possible to compute the projection of the graph at the same time as the new values for the longest paths, since the vertices are not processed in topological order. However, it is easy to compute the projection after having discovered all affected vertices without increasing the asymptotic complexity.

```
procedure insertArc($G, x \rightarrow y$)
begin
1.  $Q = \{\}$;
2.  $G = G \cup \{x \rightarrow y\}$;
3.  $\alpha(y) = l(x) + d(x,y) - l(y)$;
4.  $P = \{y\}$;
5.  if $\alpha(y) > 0$ then
6.    $insert(Q, \langle \alpha(y), y \rangle)$;
7.    while $Q \neq \emptyset$ do
8.      $u = extractMax(Q)$;
9.      $l(u) = l(u) + \alpha(u)$;
10.     forall($w \in succ(G,u)$) do
11.       $\alpha_w = l(u) + d(u,w) - l(w)$;
12.       if $\alpha_w \geq 0$ then $P = P \cup \{w\}$;
13.       if $\alpha_w > 0 \wedge (w \notin Q \vee \alpha_w > \alpha(w))$ then
14.         $\alpha(w) := \alpha_w$;
15.         $updatePriority(Q, w, \alpha_w)$;
16. forall($u \in P$) do
17.   $G_l = G_l \setminus \{x \rightarrow u \mid x \rightarrow u \in G_l\}$;
18.   $G_l = G_l \cup \{x \rightarrow u \mid x \in pred(G,u) \wedge l(x) + d(x,u) = l(u)\}$;
```

Figure 11. Algorithm for updating *l* values upon an edge insertion.

7.2. *Algorithm*

Figure 11 depicts the algorithm. It receives $G = (V, A)$, the function l for G, an arc $x \rightarrow y$ to be inserted in P and the weight $d(x, y)$ of this arc. It updates the function l for the graph $G' = (V, A \cup \{x \rightarrow y\})$.

Initially, it pushes y into the queue Q with priority equal to $\alpha(y) = \Delta(y) = l(x) + d(x, y) - l(y)$ and inserts y in the set of affected vertices P. The core of the algorithm is lines 8–15 which are iterated until the queue is empty. Each iteration extracts a vertex u with maximal priority from the queue, updates its longest path $l(u)$, and inserts it into P. At that stage, it can be proven that $\alpha(u) = \Delta(u)$ and hence $l(u)$ has now reached its final value. Lines 11–15 consider the successors of u and update their priorities if necessary. A priority is updated if the variation along the arc $u \rightarrow w$ is greater than the current priority. Note that the lengths of vertices from P cannot be updated at this point, since they have already reached final values. The algorithm terminates with a linear scan of the affected vertices to update $G_{|l}$ by removing arcs incident on affected vertices and adding back the arcs that are incident and tight.

7.3. *Proof of Correctness*

We now present the correctness proof for the insertion algorithm.

Definition 7 (Variation of a Vertex). Let $G(V, A)$, $x \rightarrow y \notin A$ and $G'(V, A \cup \{x \rightarrow y\})$. The variation of a vertex $v \in V$ from G to G', denoted by $\Delta(v, G, G')$, is defined as

$$\Delta(v, G, G') = lp(G', v) - lp(G, v).$$

In the following, we abuse notation and use $\Delta(v)$ instead of $\Delta(v, G, G')$ when G and G' are clear from the context. The next definition, the *slack* of an arc, specifies the variation that must be pushed through an arc to change the longest path of its destination.

Definition 8 (Slack of an Arc) Let $G(V, A)$ and $x \rightarrow y \in A$. The slack of *arc* $x \rightarrow y$, denoted by $sl(G, x \rightarrow y)$, is defined as

$$sl(G, x \rightarrow y) = lp(G, y) - (lp(G, x) + d(x, y)).$$

Once again, we will abuse notation by omitting G. The next proposition relates the variations of a vertex and its predecessors. It implies that the variations decrease monotonically along paths in the graph.

Proposition 4 *Let* $G(V, A)$, $x \rightarrow y \notin A$, $G'(V, A \cup \{x \rightarrow y\})$ *and* $v \in Aff_l(G, x \rightarrow y)$ $(v \neq y)$. *We have*

$$\Delta(v) = \max_{p \in pred(G,v)} (\Delta(p) - sl(p, v)).$$

Proof 8: By definition of longest paths, we have

$$\begin{aligned}
lp(G,v) &= max_{p \in pred\left(G',v\right)}(lp(G',p) + d(p,v)) \\
lp(G,v) + \Delta(v) &= max_{p \in pred(G,v)}(lp(G,p) + \Delta(p) + d(p,v)) \\
\Delta(v) &= max_{p \in pred(G,v)}(\Delta(p) - (lp(G,v) - (lp(G,p)) + d(p,v))) \\
\Delta(v) &= max_{p \in pred(G,v)}(\Delta(p) - sl(p,v))
\end{aligned}$$

■

Proposition 5 (Monotonicity). *Let $G\,(V, A)$, $x \rightarrow y \notin A$, $G'(V, A \cup \{x \rightarrow y\})$, v, $w \in Aff_I(G, x \rightarrow y)$, and $v \rightarrow w$ be an arc on a longest path in G'. We have*

$$\Delta(v) \geq \Delta(w).$$

Proof 9: Since $v \rightarrow w$ is on a longest path in G', we have

$$lp(G',w) = lp(G',v) + d(v,w).$$

Since $lp\,(G, w) \geq lp\,(G, v) + d\,(v, w)$, it follows that

$$\Delta(w) = lp(G',w) - lp(G,w) \leq lp(G',v) - lp(G,v) = \Delta(v).$$

■

We are now ready to present the correctness of Procedure `insertArc` for arcs of arbitrary weights.

Theorem 4 *Procedure `insertArc` is correct and terminates.*

Proof 10: Once again, the proof relies on the observation that the algorithm partitions the affected vertices in three sets P, Q and R

$$P = \{v \in Aff_I \mid l(v) = lp(G',v)\}; \tag{8}$$

$$Q = \{v \in Aff_I \mid \exists v' \in P : v' \rightarrow v \;\&\; v \notin P\}; \tag{9}$$

$$R = \{v \in Aff_I \mid \exists v' \in Q : aff_I{*}(v',v) \;\&\; v \notin P \cup Q\} \tag{10}$$

as well as the following invariants in line 8 of the algorithm:

$$Aff_I = P \cup Q \cup R \tag{11}$$

$$\forall w \in Q \backslash \{y\} : \alpha(w) = \max_{v \in pred(G,w) \cap P} \Delta(v) - sl(v,w) \tag{12}$$

$$\alpha(y) = lp(G,x) + d(x,y) - lp(G,y) \tag{13}$$

Initially, $P = \emptyset$, $Q = \{y\}$ and all the other affected vertices are in R, thereby establishing invariant 11. Invariants 12 and 13 trivially hold.

Assume that the invariants hold at iteration i and let us show that they also hold at the beginning of iteration $i + 1$. We first show that $\alpha(u) = \Delta(u)$ to satisfy the definition of P. It is obvious for the first iteration which extracts y. For subsequent iterations, by Proposition 4,

$$\Delta(u) = \max_{p \in pred(G,u)} \Delta(p) - sl(p,u)$$

and, by invariant 12,

$$\alpha(u) = \max_{p \in pred(G,u) \cap P} \Delta(p) - sl(p,u).$$

Since $\alpha(u) = \max_{q \in Q} \alpha(q)$, it suffices to show that

$$\forall v \in Q \cup R : \Delta(v) \leq \max_{q \in Q} \alpha(q)$$

since it implies

$$\alpha(u) \geq \max_{p \in pred(G,u) \cap (Q \cup R)} \Delta(p) \geq \max_{p \in pred(G,u) \cap (Q \cup R)} \Delta(p) - sl(p,u)$$

and the result follows by the definitions of P, Q, and R. To show

$$\forall v \in Q \cup R : \Delta(v) \leq \max_{q \in Q} \alpha(q)$$

consider the vertex $q^* \in Q \cup R$ such that

$$\Delta(q^*) = \max_{q \in Q \cup R} \Delta(q) \tag{14}$$

and such that q^* is topologically smallest among vertices satisfying 14.

Since, by definition of R, the longest paths to a vertex $r \in R$ go through vertices in Q, it follows, by Proposition 5, that $q^* \in Q$. If q^* has a predecessor w in $Q \cup R$, it follows that $\Delta(w) < \Delta(q^*)$, since q^* is topologically minimal and $\Delta(q^*)$ is maximal. It follows that the arc $w \rightarrow q^*$ cannot belong to a longest path to q^*, since this would contradict Proposition 5. As a consequence, the predecessor of q^* on any longest path must be a vertex $p^* \in P$. We have that

$$\begin{aligned}
lp(G', q^*) &= \max_{p \in pred(G', q^*) \cap P} lp(G', p) + d(p, q^*) \\
lp(G, q^*) + \Delta(q^*) &= \max_{p \in pred(G, q^*) \cap P} lp(G, p) + \Delta(P) + d(p, q^*) \\
\Delta(q^*) &= \max_{p \in pred(G, q^*) \cap P} \Delta(p) - (lp(G, q^*) - lp(G, p) + d(p, q^*)) \\
\Delta(q^*) &= \max_{p \in pred(G, q^*) \cap P} \Delta(p) - sl(p, q^*) = \alpha(q^*) \leq \max_{q \in Q} \alpha(q).
\end{aligned}$$

It remains to show that invariant 12 is preserved by lines 11–15. Consider a vertex w such that $u \rightarrow w$. The value $\alpha(w)$ becomes

$$\begin{aligned}
&max(\alpha(w), l(u) - l(w) + d(u, w)) \\
&max(\alpha(w), lp(G', u) - lp(G, w) + d(u, w)) \\
&max(\alpha(w), \Delta(u) - (lp(G, w) - (lp(G, u) + d(u, w)))) \\
&max(\alpha(w), \Delta(u) - sl(u, w)) = \max_{p \in pred(G, w) \cap P} \Delta(p) - sl(p, u).
\end{aligned}$$

The definitions of Q and R are also preserved by these instructions. Finally, the algorithm terminates, since the size of $Q \cup R$ strictly decreases at each iteration. Indeed, a vertex in P cannot be inserted in the queue more than once, since $l(u)$ reaches its final value in line 9 and, in line 12, $\alpha_w \leq 0$ for all $w \in P$. ■

7.4. Complexity

By the proof of Theorem 4, a vertex is inserted at most once into the queue. In addition, each edge outgoing from a node in Aff_I is examined once to determine whether its target should be inserted into Q (or its priority updated if it is already there). As consequence, the algorithm performs a total of $|\delta|$ insertions, $|\delta|$ extractions, and at most $\|\delta\|$ *updatePriority* operations. The total running time, using a Fibonacci heap to implement the queue, is $O(\|\delta\| + |\delta| \log|\delta|)$.

7.5. Integral Arc Weights

Whenever all the arcs in $G(V, A)$ have integral weights, the algorithm can be further improved. The basic observation is that by Proposition 5, the variations of the nodes are all integers between 0 and $\Delta(y)$, so we can use a Fibonacci heap that holds at most $\Delta(y)$ buckets, one for each possible priority. When the priority of a vertex is updated, it suffices to move it to another bucket. By implementing each bucket as a linked list, it is easy to perform these operations in constant time. Similarly, a vertex is extracted from the heap by extracting it from the largest priority bucket and removing that bucket from the heap if it becomes empty. The number of non-empty buckets is never larger than the number of items in the heap, so the runtime complexity of the algorithm drops to $O(\|\delta\| + |\delta| \log \min\{|\delta|, \Delta(y)\})$. Note that, with Thorup's integer priority queue [24], the asymptotic running time is further reduced to $O(\|\delta\| + |\delta| \log \log \min\{|\delta|, \Delta(y)\})$.

7.6. Generalizations

It is easy to generalize the insertion algorithm to insert a set of arcs of the form $\{x \to y_1, \ldots, x \to y_n\}$. Indeed, the vertex y_i with the largest priority obviously reaches its final value when extracted from the queue and the rest of the proof continues to hold. An intuitive way to understand why this holds is by looking at the transformations in Figure 12 that effectively reduce this insertion to the addition of a single 0-length arc between x and a dummy vertex z whose longest path is $-\infty$ initially, since it cannot be reached from the source. As a consequence, the addition of vertex z and the arcs $z \to y_1, \ldots, z \to y_n$ cannot affect the longest paths. The addition of the arc $x \to z$ is a single arc insertion and the algorithm described earlier applies.

The deletion procedure can be directly adapted to handle multiple arc deletions as before, since `removeArc` and `computeAffected` depend solely on the topology of the graph. Finally, the algorithms can be easily adapted to handle a single arc length change. If $d(x, y)$ for arc $x \to y$ increases, it is equivalent to the addition of a longer arc between x and y. If $d(x, y)$ for $x \to y$ decreases, it is sufficient to add a new arc between x and y with the new length $d(x, y)$, which can be done in constant time, and remove the old arc $x \to y$.

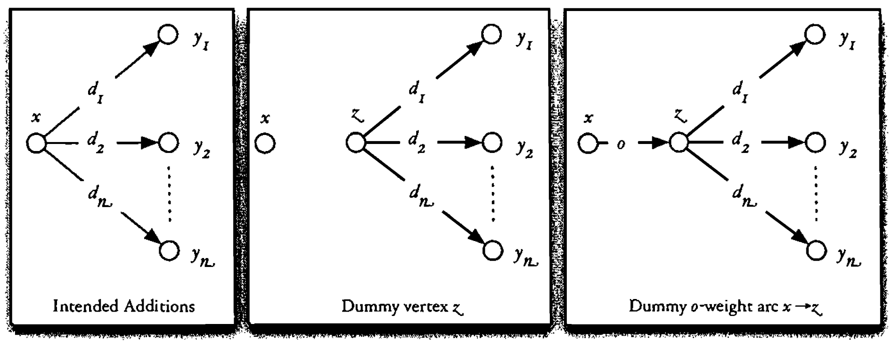

Figure 12. Transformation for addition of multiple arcs.

8. Experimental Results

Table 1 reports some preliminary experimental results on the practicality of the algorithms working with non negative arc lengths. The only purpose of these experiments is to show that the algorithms can be implemented efficiently (i.e., the constants are not prohibitive) and may bring significant benefits. To validate this claim, we instrumented an implementation of bidirectional search so that each arc addition is propagated immediately. We then compared the behavior of a differentiable object with offline and incremental algorithms. Table 1 reports the results of running the resulting procedures on 10 longest paths simultaneously to minimize the impact of other parts of the procedure. Lines `offline`, `Incr`, and `Incr(i + d + i)` describe the results for the offline implementation, the incremental implementation, and the procedure testing deletion on a 745 mhz PC. In the instrumentation `Incr(i + d + i)`, an arc addition is replaced by a sequence of three operations (addition, deletion, addition) of the same arc. Of course, the differentiable object has no idea that it is being used in a bidirectional search procedure and cannot perform any optimization. The experiments show the significant potential benefits of the incremental algorithm. For instance, `la35` shows an improvement of a factor 48 for a graph of 300 tasks. Note also the excellent times `Incr(i + d + i)`, where the times for the additional deletion and insertion are amortized by other parts of the bidirectional implementation.

9. Related Work

The bounded incremental computation (BIC) model was formally introduced by Ramalingam and Reps [19]. However, it was used as early as 1982 (by Reps again [21]) to analyze algorithms for attribute grammars, as well as in several other papers, primarily in the programming language community. Ramalingam and Reps also proposed a bounded algorithm for maintaining shortest paths, which was the inspiration for this research. Their algorithms are adaptations of Dijkstra's shortest path algorithm. Their *insertArc* procedure runs in $O(|\delta|\log|\delta| + \|\delta\|)$ time. Their *deleteArc* procedure runs in $O(|\delta|\ \log|\delta| + \|\delta\|)$ time, starts by computing the set of affected vertices using a projected subgraph, and uses the complement of the projected graph to initialize a Dijkstra-like second phase. Our deletion procedure runs in $O(\|\delta\|)$ time and uses an offline algorithm (based on degrees) on the subgraph, once the affected vertices are

Table 1. Experimental evaluation of the incremental algorithms

	abz7	abz8	abz9	la31	la32	la33	la34	la35
`offline`	88.39	87.41	87.32	157.05	159.36	156.75	166.41	155.68
`Incr`	1.93	1.94	1.94	3.40	3.44	3.39	3.45	3.45
`Incr(i + d + i)`	2.75	2.88	2.70	5.00	4.95	4.78	5.00	4.97

computed. Reference [18] presents a grammar problem which can be viewed as a generalization of the shortest path problem. When the arc weights are not negative, it is possible to reduce longest paths to this problem, since longest paths give rise to superior functions. The resulting algorithm handles arbitrary multiple insertions/deletions. However, it runs in $O(\|\delta\| \log \|\delta\|)$ time and is more costly from a practical standpoint as well. Its additional complexity is not necessary for many applications, as we discussed earlier, where our simpler algorithms are significantly faster and should be preferred. Ramalingam [20] considers incremental feasibility of systems of difference constraints using incremental shortest path algorithms. These algorithms can be applied to incremental feasibility of temporal constraint networks [4].

10. Conclusion

This paper considered invariants for longest paths in directed acyclic graphs, a fundamental abstraction for programming tools supporting local search. It presented bounded incremental algorithms for arc insertion and deletion which run in $O(|\delta| \log|\delta| + \|\delta\|)$ and $O(\|\delta\|)$ time respectively, where $|\delta|$ and $\|\delta\|$ are measures of the change in the input and output. The algorithms were also shown to be practical experimentally and their generalizations to various scheduling applications were also discussed.

There is an open issue raised by this research: It would be interesting to determine if there exists an insertion algorithm with an $O(\|\delta\|)$ time and space complexity, since the incremental algorithm has an additional log factor (in time) compared to the offline algorithm. With integer weights, the general insertion algorithm presented here can be made to run in $O(\max(\Delta(y), \|\delta\|))$ time using an array of $\Delta(y)$ buckets and $O(\max(\Delta(y), |\delta|))$ space, since the algorithm explores the buckets in decreasing order. Observe also that this implementation is not really bounded if $\Delta(y)$ is not considered part of the input, making the open issue that more relevant.

Acknowledgments

This paper is a revised and extended version of both [9, 15]. Special thanks to G. Ramalingam and Tom Reps for clarifying the links between incremental priority orderings and longest paths and to Andrei Missine and the reviewers for their comments. This work was partially supported by NSF ITRs DMI-0121495 and ACI-0121497.

Note

1. We assume that the DAG has a single source. Note, however, that a DAG with more than one source can be handled by the same algorithms by adding a new source s and an arc of weight 1 from s to each of the sources. Then, there is a path of length

k from a source of the original graph to v iff there is a path of length $k + 1$ from the source of the modified graph to v.

References

1. Aarts, E., van Laarhoven, P., Lenstra, J., & Ulder, N. (1994). A computational study of local search algorithms for job shop scheduling. *ORSA J. Comput.* 6: 113–125.
2. Alpern, B., Carle, A., Rosen, B., Sweeney, P., & Zadeck, K. (1990). Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-90)*, San Francisco, California (January).
3. Codognet, C., & Diaz, D. (2001). Yet another local search method for constraint solving. In *Proceedings of the International Symposium on Stochastic Algorithms: Foundations and Applications (SAGA 2001)*, Berlin, Germany (December), pages 73–90.
4. Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artif. Intell.* 49: 61–95.
5. Dell'Amico, M., & Trubian, M. (1993). Applying tabu search to the job-shop scheduling problem. *Ann. Oper. Res.* 41: 231–252.
6. Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.* 34: 596–615.
7. Di Gaspero, L., & Schaerf, A. (2002). *Optimization Software Class Libraries*, chapter Writing Local Search Algorithms Using EasyLocal++. Kluwer, Boston.
8. Galinier, P., & Hao, J.-K. (2000). A general approach for constraint solving by local search. In *CP-AI-OR'00*, Paderborn, Germany (March).
9. Katriel, I. (2004). Dynamic heaviest paths in DAGs with arbitrary edge weights. In *CP-AI-OR'04*, Springer Verlag, Nice, France (April).
10. Katriel, I., & Bodlaender, H. L. (2005). Online topological ordering. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2005)*, Vancouver, Canada.
11. Laburthe, F., & Caseau, Y. (1998). SALSA: A language for search algorithms. In *CP'98*, Springer Verlag, Pisa, Italy (October).
12. Michel, L., & Van Hentenryck, P. (1992). A constraint-based architecture for local search. In *OOPLSA'02*, Seattle, WA (November). ACM SIGPLAN Notices, ACM Press, New York (2002).
13. Michel, L., & Van Hentenryck, P. (2000). Localizer. *Constraints* 5: 41–82.
14. Michel, L., & Van Hentenryck, P. (2001). Localizer++: An open library for local search. Technical Report CS-01-02, Brown University.
15. Michel, L., & Van Hentenryck, P. (2003). Maintaining longest path incrementally. In *CP'03*, Kinsale, Ireland (September).
16. Michel, L., & Van Hentenryck, P. (2004). A simple tabu search for warehouse location. *Eur. J. Oper. Res.* 157/3: 576–591.
17. Nowicki, E., & Smutnicki, C. (1996). A fast taboo search algorithm for the job shop problem. *Manage. Sci.* 42(6): 797–813.
18. Ramalingam, G., & Reps, T. (1996). An incremental algorithm for a generalization of the shortest-path problem. *J. Algorithms* 21: 267–305.
19. Ramalingam, G., & Reps, T. (1996). On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.* 158: 233–277.
20. Ramalingam, G., Song, J., Joscovicz, L., & Miller, R. E. (1999). Solving difference constraints incrementally. *Algorithmica* 23: 261–275.
21. Reps, T. (1982). Optimal-time incremental semantic analysis for syntax-directed editors. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages (POPL-82)*, ACM Press.
22. Shaw, P., De Backer, B., & Furnon, V. (2002). Improved local search for CP toolkits. *Ann. Oper. Res.* 115: 31–50.
23. Tarjan, R. (1985). Amortized computational complexity. *SIAM J. Algebr. Discrete Methods* 6: 306–318.

24. Thorup, M. (2003). Integer priority queues with decrease key in constant time and the single source shortest paths problems. In *Proc. of the 35th ACM Symp. on Theory of Computing (STOC)*, pages 149–158.
25. Voss, S., & Woodruff, D. (2002). *Optimization Software Class Libraries*. Kluwer, Boston.
26. Voudouris, C., Dorne, R., Lesaint, D., & Liret, A. (2001). iOpt: A software toolkit for heuristic search methods. In *CP'01*, Springer Verlag, Paphos, Cyprus (October).
27. Walser, J. (1998). *Integer Optimization by Local Search*. Springer Verlag, Berlin.
28. Werner, F., & Winkler, A. (1995). Insertion techniques for the heuristic solution of the job-shop problem. *Discrete Appl. Math.* 58(2): 191–211.