Maintaining Longest Paths Incrementally

Laurent Michel[1] and Pascal Van Hentenryck[2]

[1] University of Connecticut, Storrs, CT 06269-3155
[2] Brown University, Box 1910, Providence, RI 02912

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 $O(\|\delta\| log \|\delta\|)$ and $O(\|\delta\|)$ respectively, where $\|\delta\|$ is a measure 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.

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, [6,8,10,18,20,22]).

The design of modeling and programming tools for local search generally involves abstractions to express the neighborhood and to encapsulate incremental algorithms. Localizer [10] 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 [8,21]. More recently, constraint-based approaches to local search (e.g., [3,7,11,22]) were proposed, where constraints incrementally maintain properties such as their violation degrees. The Comet system [9] 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 [9,12].

This paper was motivated by the study of differentiable objects for scheduling applications, where it is often critical to maintain longest paths in directed acyclic graphs (DAG) 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., [23]), as well as in neighborhood search (e.g., [1,5,13]). 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.

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 $O(\|\delta\| log \|\delta\|)$ (insertion) and $O(|\delta\|)$ (deletion), where $\|\delta\|$ represents the *size of the changes in the input and output*[1]. The results use the Bounded Incremental Computation (BIC) model of Ramalingam and Reps [15]. 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 detail and give their correctness proofs. Section 6 presents generalizations to the algorithms, as well as their applications to scheduling. Section 7 gives some preliminary experimental results, Section 8 describes related work, while Section 9 concludes the paper.

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 on 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 paths from the source to all 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 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 (e.g., [19]), and *bounded incremental computation* (BIC) [15]. Many such models analyze the complexity

[1] We give more precise bounds later in the paper when the terminology is introduced.

of incremental algorithm 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 both 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 remain 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 (vertices,lenghts) 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 all allowed change ϵ, its execution time depends only on δ, not the size of the entire input $x + \epsilon$. It is *unbounded* otherwise. Of course, many incremental algorithms are unbounded (e.g., graph reachability) and hence the existence of a bounded algorithm is a strong guarantee for incremental performance.

An example of bounded incremental algorithm is the shortest path algorithm of Ramalingam and Reps [15], which runs in $O(\|\delta\| \; log\|\delta\|)$ for arc insertions and deletions, when the arc weights are strictly positive. Here $\|\delta\|$ denotes the number of *affected vertices*, i.e., the vertices whose shortest paths have changed, and their adjacent arcs. It is natural to use $\|\delta\|$, and not $|\delta|$, since any algorithm would necessarily have to examine the adjacent vertices 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 for incremental longest paths in a DAG. The algorithm takes $O(|\delta| \; log|\delta| + \|\delta\|)$ for an arc insertion and $O(\|\delta\|)$ for 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

We now give the high-level intuition behind the algorithms presented in this paper and we explain why some simple and natural ideas do not lead to bounded algorithms. We initially focus on graphs with strictly positive weights. This restriction is lifted in Section 6. Throughout the paper, we use directed acyclic graphs with a source s. 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

```
1.  forall($v \in V$) do
2.       $degree(v) = |pred(G, v)|$;
3.  $Q = \{v \mid degree(v) = 0\}$;
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.            $degree(w) = degree(w) - 1$;
9.            if $degree(w) = 0$ then
10.                $insert(Q, w)$;
```

Fig. 1. An Offline Algorithm for Longest Path in a DAG.

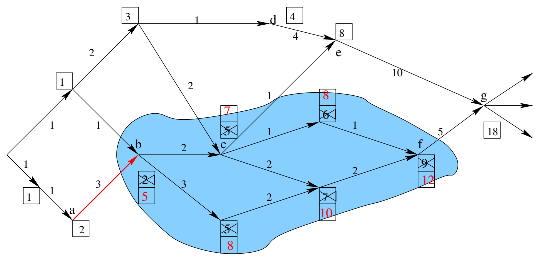

Fig. 2. The Affected Set of an Insertion.

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

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

Figure 1 presents an offline algorithm for longest paths in a DAG, which runs in $O(|V| + |E|)$ for a directed acyclic graph $G = (V, E)$. The key idea of the algorithm is to consider the vertices in topological order, which guarantees that, when a vertex is dequeued, its predecessors have the correct longest path values. Lines 1-2 compute the initial degree of the vertices and Line 3 inserts the source in the queue. Lines 4 and 5 dequeue a vertex and compute the length of its longest path from the source. Lines 7 to 9 decrease the degrees of the successors of v and insert them in the queue if all their predecessors have been updated, i.e., when their degrees is 0.

Consider now the problem of updating the longest paths after 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 coming from f is not longer than the longest path from e.

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

```
procedure insertArc(G,x → 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
```

Fig. 3. A Preliminary Version of Procedure `insertArc`.

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 degrees in the offline algorithm is, in fact, a simple way to order the vertices topologically. This approach is appealling, since there exists a bounded incremental algorithm for priority ordering which can be used for that purpose [2]. 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 $a \to b$ 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 and to enqueue the successors of affected vertices when the lengths of their longest paths are increasing. Such an algorithm is shown in Figure 3. Let G^- be the graph G at call time. Line 2 tests whether the new arc $x \to y$ changes the longest path of its destination y. If it does, then y is inserted in the queue with $l(y)$, its longest path in G^-, as its key. The affected vertices are computed and processed in lines 5-9. Line 5 pops the vertex v with the smallest key and updates its longest paths. It then considers each successor w of v and inserts w in the queue if its longest paths 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 updates a key in the queue) and each affected vertex enters the queue at most 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.

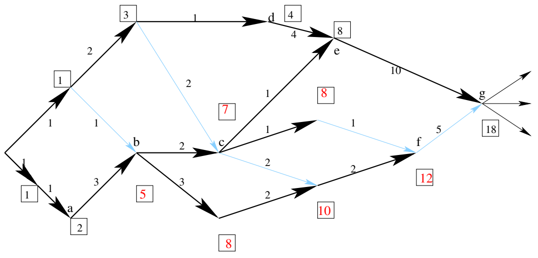

Fig. 4. The Longest Path Projection $G_{|l}$.

This is possible because it is sufficient to notice that the length of a longest path decreases: it is not necessary to know by how much. More precisely, arc deletion can be tought of as working on subgraphs $G_{|l}$ obtained by 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 incident to w in $G_{|l}$. By proceeding this way, all affected vertices can be computed in $O(\|\delta\|)$. 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 incident arc in $G_{|l}$. On the other hand, vertex e is not affected since it has two incident arcs. 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 degrees 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 weigths, this restriction being lifted in Section 6.

4 Insertion

Figure 5 depicts procedure `insertArc`. The main differences with the preliminary version presented earlier are lines 7-8 and 12-15, which maintain the projected graph. Lines 7-8 updates the projected graph for an affected vertex v, lines 12-13 adds an arc originating from an affected vertex to a non-affected vertex, while lines 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 $G = (V, A)$, $x \to y \notin A$, and $G' = (V, A \cup \{x \to y\})$. The set of affected vertices by the insertion of $x \to y$ in G is defined as*

$$AffectedI(G, x \to y) = \{v \in V \mid lp(G', v) > lp(G, v)\}.$$

```
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
```

Fig. 5. Procedure `insertArc`.

In the following, we abuse notations and remove the arguments of *AffectedI* when they are clear from the context. The following proposition informally states that a vertex is affected only if one of its predecessors is affected.

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

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

The proposition makes it natural to define a binary relation *affectI*.

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

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

We use affectI to denote the transitive closure of affectI.*

The following proposition characterizes the affected vertices.

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

Definition 3 (Specification of `insertArc`). *Let $G = (V, A)$ be a DAG with strictly positive 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. The proof relies on the observation that the algorithm partitions the affected vertices in three sets

$$\begin{aligned}
P &= \{x \in \mathit{AffectedI} \mid l(x) = lp(G', x)\};\\
Q &= \{x \in \mathit{AffectedI} \mid \exists v \in P : v \to x \;\&\; x \notin P\};\\
R &= \{x \in \mathit{AffectedI} \mid \exists v \in Q : \mathit{affectI}^*(v, x) \;\&\; x \notin P \cup Q\}
\end{aligned}$$

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

$$\begin{aligned}
&\mathit{AffectedI} = P \cup Q \cup R && (1)\\
&\forall v \in P, \forall x \in Q : lp(G, v) \leq lp(G, x). && (2)
\end{aligned}$$

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

$$\forall y \in \mathit{succ}(G, x) : lp(G, x) < lp(G, y).$$

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 restored. By selection of v and since $\forall y \in \mathit{succ}(G, x) : lp(G, x) < lp(G, y)$, 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.

5 Arc Deletion

Figures 6 and 7 depict the algorithms to compute the deletion of an arc $x \to y$. Function `computeAffected` in Figure 6 computes the set of affected vertices by a deletion. It starts with the deleted arc $x \to y$ 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 its predecessors in the projected graph are affected. This is tested by removing from G_l all arcs $v \to w$, 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 $x \to y$ affects y, the procedure computes the affected vertices using function `computeAffected`. It then initializes the degrees 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 traditional offline algorithm on the affected vertices. We now formalize the various concepts and give the correctness proofs.

```
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.          $G_l = G_l \setminus \{u \to v\}$;
8.          if $pred(G_l, v) = \emptyset$ then
9.              $insert(Q, v)$;
10. return $A$;
end
```

Fig. 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.              $degreelp(v) = |pred(G, v) \cap Affected|$;
8.          $Q = \{v \in Affected \mid degreelp(v) = 0\}$;
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.                 $degreelp(w) = degreelp(w) - 1$;
15.                 if $degreelp(w) = 0$ then $insert(Q, w)$;
end
```

Fig. 7. Procedure `removeArc`.

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

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

As before, we abuse notations and remove the arguments of *AffectedD* 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** 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*

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

Proof. 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 $\forall p \in pred(G_{|l}, v) : lp(G', p) < lp(G, p)$. The result follows.

Corollary 1. *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 implies $y \to_l^* v$.*

Proof. 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 a DAG with strictly positive weights, $x \to y \in A$, $G' = (V, A \setminus \{x \to y\})$, and $lp(G', y) < lp(G, y)$. Procedure* `computeAffected(`$G, x \to y$`)` *satisfies the specification:*

Pre: $G_l = G_{|l}$.
Post: $G_l = G'_{|l} \setminus \{v \to w \ \mid \ v \in AffectedD\}$;
the function returns $AffectedD$.

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

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

$$\begin{array}{ll}
v \in A \Rightarrow v \in AffectedD & (1)\\
v \in Q \Rightarrow v \in AffectedD & (2)\\
R = \{w \in AffectedD \setminus (A \cup Q) \ \mid \ \exists v \in Q : v \to_l^* v\} & (3)\\
G_l = G_l \setminus \{v \to w | v \in A \cup Q\} & (4).
\end{array}$$

in line 3 of the algorithm. Initially, A is empty, $Q = \{y\}$, and the invariants hold by Corollary 1. By Invariant (2), lines 4 and 5 are correct. Moreover, if v is a successor of u and the test on line 8 succeeds, by Invariant (4), all predecessors of v must be in $A \cup Q$ 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 $A \cup Q$ 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 A 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 a DAG with strictly positive weights, $x \to y \in A$, and $G' = (V, A \setminus \{x \to y\})$. Procedure* `removeArc(`$G, x \to y$`)` *satisfies the specification:*

```
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
```

Fig. 8. Procedure `propagateChanges`.

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. The proof follows from Theorem 2 and the fact that the degrees for the non-affected vertices are initialized correctly.

6 Generalizations and Applications to Scheduling

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 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

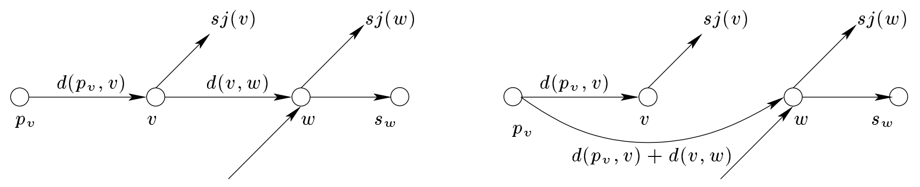

Fig. 9. Inverting Two Vertices on a Critical Path.

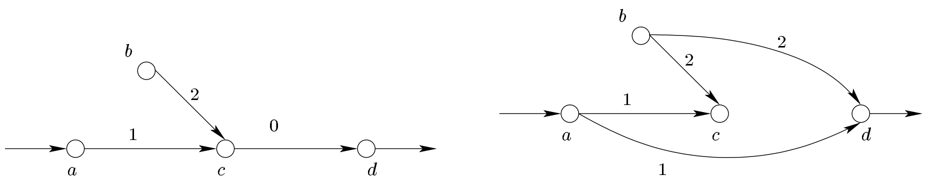

Fig. 10. A Graph with a Zero-Weight Arc and its Transformation.

algorithm of Nowicki [13]. 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 $p_v \to w$ with weight $d(p_v, v) + d(v, w)$ in constant time, since no vertex is affected. We can now remove $v \to w$ 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 $p_v \to v$ and $p_v \to w$ 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 [23] for scheduling. Observe also that our deletion algorithm supports multiple deletion naturally, since it only reasons on the projected graph.

Zero Weight Arcs. Our algorithm naturally generalizes to the case of zero-weight arcs. The difficulty here is that several vertices may have the same longest path lengths, although they are topologically ordered. Consider, for instance, the left side of Figure 10 which depicts parts of a DAG and assume that vertices c and d have the same longest path lengths and are affected (due to some of their predecessors). Vertices c and d are thus on the queue and d could be dequeued before c, although it comes after c in the topological ordering. This does not raise any major issue however. The intuition is to recognize that the arc $c \to d$ can be replaced by adding arcs $p \to d$ for each arc $p \to c$, and that this transformation, whose result is shown in on the right side of Figure 10, preserves the longest paths. After the transformation, observe that c and d are topologically independent and can be processed in any order.

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

Negative Weight Arcs. Negative weights can be handled by a similar transformation. When an arc $a \to b$ has a negative weight, it must be replaced by arcs of the form $p \to b$ for each predecessor p of a, whose weights must be reduced appropriately. In scheduling applications, these negative arcs represent a generalization of precedence constraints: they are not dynamic and generally shorter than the duration of the tasks. Hence the transformation is simple and only introduce a marginal increase in the size of the graph. Even if such insertions are dynamic, they correspond to cases which are well-handled by our algorithm, since they preserve the existing topological order of the affected vertices. The bookkeeping is however more tedious, since a more complex mapping between actual and virtual arcs must be maintained.

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

7 Experimental Results

Table 1 reports some preliminary experimental results on the practicality of the algorithms. 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. Line `offline` depicts the offline implementation, line `Incr` gives the results of the incremental implementation, and line `Incr(i+d+i)` describes the results of the procedure testing deletion. In the instrumentation `Incr(i+d+i)`, an arc addition is replaced by a sequence of three operations (addition,deletion,addition) of the same arcs. Of course, the differentiable object has no idea that it is being used in a bidirectional search procedure and cannot perform any optimization. The results show the significant benefits that may result from 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.

8 Related Work

The bounded incremental computation (BID) model was formally introduced by Ramalingam and Reps [15]. However, it was used as early as 1982 (by Reps again [17]) 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, while ours are adaptations of topological sorting for longest paths. Their *insertArc* procedure runs in $O(|\delta| log|\delta| + \|\delta\|)$, but it needs a Fibonacci heap, since it updates elements of the queue. Their *deleteArc* procedure runs in $O(|\delta| log|\delta| + \|\delta\|)$, starts by computing the set of affected vertices using a projected subgraph, and uses the completement of the projected graph to initialize a Dijkstra-like second phase. Our deletion procedure runs in $O(\|\delta\|)$ and uses an offline algorithm (based on degrees) on the subgraph, once the affected vertices are computed. Reference [14] presents a grammar problem which can be viewed as a generalization of the shortest path problem. Using the transformations described earlier, 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 $O(\|\delta\| log\|\delta\|)$ 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 [16] 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].

9 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 $O(\|\delta\|)$ respectively, where $\|\delta\|$ is a measure 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 are several open issues raised by this research. On the one hand, it would be interesting to determine if there exists a $O(\|\delta\|)$ insertion algorithm, since the incremental algorithm has an additional log factor compared to the offline algorithm. On the other hand, it would be interesting to find out an algorithm that can handle negative weights without graph transformations.

Acknowledgments

Special thanks to G. Ramalingam and Tom Reps for clarifying the links between incremental priority orderings and longest paths. This work was partially supported by NSF ITRs DMI-0121495 and ACI-0121497.

References

1. E. Aarts, P. van Laarhoven, J. Lenstra, and N. Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing*, 6:113–125, 1994.
2. B. Alpern, R. Hoover, B. Rosen, P. Sweeney, and K. Zadeck. Incremental Evaluation of Computational Circuits. In *SODA-90*, 1990.
3. C. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. In *AAAI Fall Symposium on Using Uncertainty within Computation*, Cape Cod, MA., 2001.
4. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. In *KR-89*, 1989.
5. M. Dell'Amico and M. Trubian. Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252, 1993.
6. L. Di Gaspero and A. Schaerf. *Optimization Software Class Libraries*, chapter Writing Local Search Algorithms Using EasyLocal++. Kluwer, 2002.
7. P. Galinier and J.-K. Hao. A General Approach for Constraint Solving by Local Search. In *CP-AI-OR'00*, Paderborn, Germany, March 2000.
8. F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *CP'98)*, Pisa, Italy, October 1998.
9. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *OOPLSA'02*, Seattle, WA, November 1992.
10. L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.
11. L. Michel and P. Van Hentenryck. Localizer++: An Open Library for Local Search. Technical Report CS-01-02, Brown University, 2001.
12. L. Michel and P. Van Hentenryck. A simple tabu search for warehouse location. *European Journal on Operations Research*, 2001. (to appear).
13. E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.
14. G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21:267–305, 1996.
15. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.
16. G. Ramalingam, J. Song, L. Joscovicz, and R. E. Miller. Solving difference constraints incrementally. *Algorithmica*, 23:261–275, 1999.
17. T. Reps. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. In *POPL-82*, 1982.
18. P. Shaw, B. De Backer, and V. Furnon. Improved local search for CP toolkits. *Annals of Operations Research*, 115:31–50, 2002.
19. R. Tarjan. Amortized Computational Complexity. *SIAM Journal of Algebraic Discrete Methods*, 6:306–318, 1985.
20. S. Voss and D. Woodruff. *Optimization Software Class Libraries*. Kluwer, 2002.
21. C. Voudouris, R. Dorne, D. Lesaint, and A. Liret. iOpt: A Software Toolkit for Heuristic Search Methods. In *CP'01*, Paphos, Cyprus, October 2001.
22. J. Walser. *Integer Optimization by Local Search*. Springer Verlag, 1998.
23. F. Werner and A. Winkler. Insertion techniques for the heuristic solution of the job-shop problem. *Discrete Applied Mathematics*, 58(2):191–211, 1995.