On competitive on-line algorithms for the dynamic priority-ordering problem

G. Ramalingam and Thomas Reps

Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706 USA

Abstract

Ramalingam, G. and Reps, T., On competitive on-line algorithms for the dynamic priority-ordering problem.

The vertices of a directed acyclic graph (DAG) are said to be *correctly prioritized* if every vertex v in the graph is assigned a priority, denoted by $priority(v)$, such that if there is an edge in the DAG from vertex v to vertex w then $priority(v) < priority(w)$. The dynamic priority-ordering problem is to maintain a correct prioritization of the graph as the DAG is modified. Alpern *et al.* presented an algorithm for this problem. In this paper we show that the Alpern *et al.* algorithm does not have a constant *competitive ratio*, where the cost of the algorithm is measured in terms of the number of primitive priority-manipulation operations. The proof also shows that there exists no algorithm for the problem that has a constant *competitive ratio*, as long as the allowed primitive priority-manipulation operations satisfy a simple property. The proof that we give also shows that there exists no algorithm for the problem of maintaining a topological-sort ordering that has a constant competitive ratio.

Keywords: Analysis of algorithms, competitive ratio, dynamic priority-ordering problem, dynamic topological-sorting problem

1. Introduction

The vertices of a directed acyclic graph (DAG) are said to be *correctly prioritized* if every vertex v in the graph is assigned a priority, denoted by $priority(v)$ and drawn, say, from the non-negative integers, such that if there is an edge in the graph from vertex v to vertex w then $priority(v) < priority(w)$. (Note that a given DAG has infinitely many correct prioritizations.) The *dynamic priority-ordering problem* is to maintain a correct prioritization of the DAG as it is modified by inserting or deleting vertices and edges [5,2].

Previous work has addressed two versions of the problem. Hoover gave an algorithm for the restricted version of the problem where each modification of the DAG consists of a *unit change* [5, pp. 19-23]. In this version of the problem, each modification to the DAG involves the insertion or deletion of exactly *one* edge, after which the DAG's priorities are updated so that the vertices are again correctly prioritized. Although restricted to unit changes, Hoover's method

---

Correspondence to: T. Reps, Computer Sciences Department, University of Wisconsin–Madison, 1210 West Dayton Street, Madison, WI 53706 USA. Email: reps@cs.wisc.edu.

addresses the fully dynamic version of the problem in that a mixture of insertions and deletions are permitted in a sequence of graph modifications. Alpern *et al.* gave an algorithm allowing multiple, heterogeneous changes: between updates, the DAG is allowed to be restructured by an arbitrary mixture of insertions and deletions [2].

In this paper, we address the question of how well an algorithm for the dynamic priority-ordering problem can perform as an on-line algorithm responding to a sequence of requests of two types: (i) requests to perform unit-size graph-modification operations, and (ii) requests to update priority values. The answer we obtain for this question applies equally well to both versions of the problem described above: the unit-change version corresponds to requiring the request sequence to have an update request following each (unit-size) graph-modification request; the multiple-heterogeneous-change version corresponds to allowing an arbitrary collection of graph-modification requests between update requests.

The notion of an on-line algorithm's *competitive ratio* has been proposed as a way to measure the performance of algorithms over a sequence of operations [16,8,6]. An on-line algorithm receives a sequence of requests, and after each request it responds with a corresponding action. Each action has an associated cost, and the cost of the algorithm for the request sequence is the sum of costs of the actions. An off-line algorithm for the same problem is permitted to examine the entire sequence of requests before choosing its actions. To determine an on-line algorithm's competitive ratio, one assumes that there is an adversary with "maximally destructive intent" generating the requests. The notion of competitiveness assesses the amount of "damage" that the adversary can inflict, in the sense that the competitive ratio compares the performance of the on-line algorithm to the performance of an optimal algorithm for the off-line version of the problem. The competitive ratio is the maximum value—over any sequence of requests—of the ratio between the cost of the on-line algorithm and the cost of an optimal off-line algorithm. Thus, an algorithm designer seeks a competitive ratio as small as possible (where the smallest possible ratio is 1).

In this paper, we show that the algorithm presented by Alpern *et al.* for the dynamic priority-ordering problem does not have a constant competitive ratio. The proof also shows that no algorithm for the dynamic priority-ordering problem can have a constant competitive ratio as long as the primitive priority-manipulation operations used by the algorithm satisfy a simple property. Consider the abstract interface of the set of priority-manipulation operations. This interface will

include an operation for comparing two priorities, operations for creating or generating new priorities (for example, an operation *create_between*(p, q) might generate a new priority that lies between the two specified priorities), and possibly other operations with priorities. Let us say that a particular implementation of a priority space has a *functional* interface if the execution of an operation *op(*$p_1, \ldots, p_k$*)* does not have the side effect of changing the order between two priorities q_1 and q_2 if $\{q_1, q_2\} \cap \{p_1, \ldots, p_k\} = \emptyset$. We show that no algorithm that makes use of a priority space with a functional interface can have a constant competitive ratio. (We measure the cost of the algorithm's actions in terms of the number of priority-space operations and priority assignments made.) The proof that we give also shows that there exists no algorithm for the dynamic version of the topological-sorting problem [7, pp. 258-268]—*i.e.*, the problem of maintaining a topological-sort ordering of the vertices of a DAG as the DAG undergoes changes—that has a constant competitive ratio.

The remainder of the paper consists of two sections. Section 2 discusses two issues that motivated the question addressed in the paper. Section 3 presents the proof that there exists no algorithm for the dynamic priority-ordering problem that has a constant competitive ratio, under the above assumptions.

2. Motivation and relationship with previous work

The dynamic circuit-annotation problem

Our work was partly motivated by the possibility of using an algorithm for the dynamic priority-ordering problem as a subroutine in an algorithm for the *dynamic circuit-annotation problem* [2]. A *circuit* is a DAG where every vertex u is associated with a function F_u. The output value to be computed at any vertex u is obtained by applying function F_u to the values computed at the predecessors of vertex u. The circuit-annotation problem is to compute the output value associated with each vertex. The dynamic version of the problem is to maintain consistent values at each vertex as the circuit undergoes changes [9,13,5,1,2,10].

From a systems-building perspective, the dynamic circuit-annotation problem is important because it is at the heart of several important kinds of interactive systems, including the all-pervasive spreadsheet [3,9] as well as language-sensitive editors created from attribute-grammar specifications [15]. In the case of interactive systems based on attribute grammars, specialized algorithms have been devised that take advantage of the special structure of the problem

[13,14,17,15]. However, a generalized framework has been proposed by Alpern *et al.* that uses the annotation of graphs as a paradigm for specifying other classes of interactive systems, especially ones that cannot be encoded efficiently with attribute grammars [1]. Systems created using this paradigm can give rise to arbitrary circuits. Thus, the dynamic circuit-annotation problem is highly relevant to real-world systems.

It is useful to identify two classes of algorithms for the dynamic circuit-annotation problem: *conservative* algorithms and *speculative* algorithms. Conservative update algorithms are based on the following observations:

> If during the process of assigning new values, a vertex is ever (temporarily) assigned a value other than its correct final value, spurious changes are apt to propagate arbitrarily far in the DAG. To avoid this possibility, an updating algorithm should schedule vertices for re-evaluation such that any new value computed is necessarily the correct final value. That is, a vertex should not be re-evaluated until all of its arguments are known to have their correct final values.

In contrast, speculative update algorithms may temporarily assign a vertex a value other than its correct final value. Speculative algorithms for the dynamic circuit-annotation problem have been given by Ramalingam and Reps [10,11].

A conservative update strategy, however, is possible only if the updating algorithm has some information about the topological structure of the circuit. In [2], Alpern *et al.* proposed the following algorithm for the dynamic circuit-annotation problem:

> In addition to maintaining the values that annotate the vertices of a circuit, the algorithm maintains a correct prioritization of the circuit's vertices. After a circuit is modified, first the priorities are updated so that the vertices are again correctly prioritized, and then—using the updated priorities to schedule vertices for re-evaluation—the values are updated. The re-evaluation phase makes use of a worklist—implemented as a priority queue—to keep track of all vertices for which at least one predecessor has changed value. (The worklist is initialized with all vertices of the circuit whose vertex-definition function was altered.) At each step, the vertex selected for re-evaluation is one with minimum priority value, and consequently the order in which vertices are re-evaluated is one that respects a topological-sort order of the circuit.

The relevance of the dynamic priority-ordering problem to the dynamic circuit-annotation problem directly motivates the issue of trying to characterize how good the Alpern *et al.* priority-ordering algorithm is.

Analysis of dynamic priority-ordering over a sequence of operations

A common way to evaluate the time complexity of an algorithm or problem is to use asymptotic worst-case analysis and to express the cost of the computation as a function of the size of the input. However, this approach to algorithm analysis fails to distinguish between any dynamic algorithm for the priority-ordering problem and the "start-over" algorithm that discards all priorities and re-assigns priorities from scratch. As we will see in Section 3, for any dynamic priority-ordering algorithm there are examples for which a single change to the DAG forces the algorithm to give new priorities to $\Omega(n)$ vertices.

Because an algorithm for a dynamic problem makes use of the solution to one problem instance to find the solution to a "nearby" problem instance, an alternative way to measure an algorithm's cost is to measure the time complexity of a dynamic algorithm in terms of the sum of the sizes of the *changes* in the input and output [13,2,10,12]. A dynamic algorithm is said to be *bounded* if, for all input data-sets and for all changes that can be applied to an input data-set, the time it takes to update the output solution depends only on the size of the *change* in the input and output, and not on the size of the *entire* current input. Otherwise, a dynamic algorithm is said to be *unbounded*. A dynamic problem is said to be bounded (unbounded) if it has (does not have) a bounded algorithm.

More precisely, "boundedness analysis" measures the cost of an algorithm for a dynamic problem in terms of a parameter $\|\delta\|$ that reflects the size of the change in the input and output. For example, in the dynamic priority-ordering problem, for a given modification δ, the quantity $\|\delta\|$ is defined in terms of a set S_δ consisting of the minimum number of vertices whose priority needs to be changed to have a correct prioritization of the modified graph; $\|\delta\|$ is the sum of the number of vertices in S_δ plus the number of edges with at least one endpoint in S_δ. (For a formal definition of $\|\delta\|$, see [2,10,12].)

Alpern *et al.* presented an algorithm that, after the insertion and/or deletion of (possibly many) edges from a prioritized DAG, computes a correct prioritization of the new DAG by reassigning new priorities to only $O(\|\delta\|)$ vertices.[1] In other words, the number of priority re-assignments that

[1]The priority-updating algorithm itself runs in time $O(\|\delta\|^2 \log \|\delta\|)$ [2]. Furthermore, when each modification of the DAG consists of a unit change (*i.e.*, the insertion or deletion of a single edge), the algorithm runs in time $O(\|\delta\| \log \|\delta\|)$.

the algorithm performs is at most a constant factor times the number of priority re-assignments that any algorithm for the problem must make (for bounded-degree graphs).

However, unlike some other graph-annotation problems, such as the single-source shortest-path problem, where there is *one* solution for a given graph, the priority-ordering problem admits *multiple* solutions. A given graph has *infinitely many* correct prioritizations. As defined by Alpern *et al.*, the quantity $\|\delta\|$ used in the analysis of their priority-ordering algorithm is defined in terms of the solution closest to the previous solution: $\|\delta\|$ is related to the size of the *minimal change to the current prioritization* needed to reach any of the correct prioritizations for the modified graph. Consequently, this analysis tells us nothing about the behavior of the algorithm over a sequence of operations. In particular, it does not tell us if, over a sequence of operations, the Alpern *et al.* algorithm performs at most a constant factor times the number of priority re-assignments that any algorithm for the problem must make.

This raised the question "Does the Alpern *et al.* algorithm have a constant competitive ratio?" The material presented in Section 3 shows that the answer to this question is: "The algorithm does not have a constant competitive ratio, but neither does any other on-line priority-ordering algorithm that uses a priority space with a functional interface."

3. The absence of a competitive algorithm for the dynamic priority-ordering problem

We show that the Alpern *et al.* algorithm for the dynamic priority-ordering problem is not competitive by presenting a sequence of edge insertions for which the ratio of the number of priority reassignments made by the algorithm to the number of priority reassignments made by a simple offline algorithm is not bounded by a constant.

As explained in the introduction, the proof we present in this section applies to any algorithm that makes use of a priority space with a functional interface. The Alpern *et al.* algorithm makes use of a data structure developed by Dietz and Sleator that implements a densely-ordered priority space [4]. This data structure has the following interface:

NextAfter(r):	Return a priority value q, not previously in use, such that $q > r$ and for all p in use such that $p > r$, $p > q$.
Delete(p):	Remove p from the set of priority values in use
Order(p, q):	Test whether $p < q$
$-\infty, \infty$:	constant priorities

Note that none of the above priority-space operations changes the order between any two priorities already in use. We say that a particular implementation of a priority space has *functional* interface if the execution of an operation *op*$(p_1, \ldots, p_k)$ does not have the side effect of changing the order between two priorities q_1 and q_2 if $\{q_1, q_2\} \cap \{p_1, \ldots, p_k\} = \emptyset$. We now show that no algorithm that makes use of a priority space with a functional interface can have a constant competitive ratio. (We measure the cost of the algorithm's actions in terms of the number of priority-space operations and priority assignments made.)

When we refer to any on-line algorithm in this section, it will be assumed to be one that makes use of a priority space with a functional interface. An algorithm will typically update the priority-ordering by "changing" the priorities of some particular set of vertices. The priority of a vertex may be "changed" by either explicitly assigning a new priority to that vertex or by invoking a priority-space operation to change the relative position of the particular priority in the priority space. When we talk of a priority re-assignment in this section, we include both these ways of changing a vertex's priority.

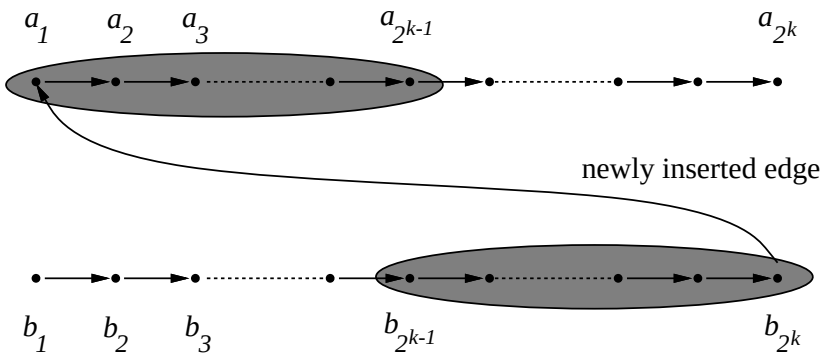
We will consider a class of problem instances of the following kind: initially the DAG will consist of just n isolated vertices; the input sequence will consist of $n-1$ edge-insertion requests that convert this DAG into a chain of n vertices. Note that there is a simple off-line algorithm that is optimal for the problem instances in this class: the edge-insertion operations are processed to create the chain; the chain is then traversed starting at its root and the vertices are assigned priorities in the sequence from 1 to n, in the order encountered. Because the only requests are edge-insertion operations, the final prioritization is a correct prioritization of the graph's vertices at any *intermediate* stage in the request sequence. This off-line algorithm makes $\Theta(n)$ priority assignments (and has a total running time of $\Theta(n)$).

Consider a specific on-line priority-ordering algorithm. We will show by induction on k that there exists an input sequence of $2^k - 1$ edge insertions that creates a chain of 2^k vertices for which the on-line algorithm must have performed at least $(k-1)2^{k-2}$ priority re-assignments. It will follow from this that the competitive ratio of the on-line algorithm must be $\Omega(\log_2 n)$, where n denotes the input size ($n = O(2^k)$).

For $k = 1$, the result follows trivially, since $(k-1)2^{k-2}$ is zero. Assume that the result holds true for a specific k. We show that it holds true for $k+1$, too. We start with 2^{k+1} isolated vertices, which we split into 2 groups of 2^k vertices each. For each of these two groups of vertices

we insert $2^k - 1$ edges that make the on-line algorithm perform at least $(k-1)2^{k-2}$ priority re-assignments. (We know that such a sequence of edge insertions exists from the inductive hypothesis.)

At this point, we have two chains, A and B, each of length 2^k, and both the chains are correctly prioritized. Let a_i and b_i denote the i-th element in the chains A and B, respectively. See the figure below.



Assume without loss of generality that $priority(a_{2^{k-1}}) \leq priority(b_{2^{k-1}})$. (If not, exchange the roles of the chains A and B in what follows.) Since priorities increase within each chain, the priorities of every vertex in the first half of chain A (shown shaded in the above picture) must be less than or equal to the priority of every vertex in the last half of chain B (shown shaded in the above picture). Now insert an edge from b_{2^k} to a_1. The algorithm must re-assign the priorities of every vertex in either the first half of chain A or the second half of chain B—otherwise, we would end up with a vertex a_i in the first half of A and a vertex b_j in the second half of B such that $priority(a_i) \leq priority(b_j)$. In other words, the algorithm must re-assign the priorities of at least 2^{k-1} vertices.

Thus, over the sequence of $2^{k+1} - 1$ edge insertions the algorithm performs at least $2 \times (k-1)2^{(k-2)} + 2^{k-1} = k2^{k-1}$ priority re-assignments. Thus, it follows that the inductive claim holds true for $k+1$, too.

It follows from this argument that when we restrict attention to the class of problems described above, the off-line algorithm performs $\Theta(n)$ priority assignments, while, in the worst case, any on-line priority-ordering algorithm must perform $\Omega(n \log_2 n)$ priority-assignment operations. Thus, the competitive ratio of any on-line priority-ordering algorithm must be $\Omega(\log_2 n)$. From this we conclude the following:

Theorem. *There exists no algorithm, utilizing a functional priority space, for the dynamic priority-ordering problem that has a constant competitive ratio.*

Note that it follows immediately from the above proof that there exists no algorithm for the dynamic topological-sorting problem that has a constant competitive ratio: (i) the optimal off-line algorithm produces a topological-sort ordering of the vertices of the DAG in time $\Theta(n)$; (ii) a topological-sort ordering of the vertices of a DAG is also a correct prioritization; hence, in the worst case, the argument given above shows that any on-line topological-sorting algorithm must perform $\Omega(n \log_2 n)$ priority-assignment operations.

Corollary. *There exists no algorithm for the dynamic topological-sorting problem that has a constant competitive ratio.*

References

1. Alpern, B., Carle, A., Rosen, B., Sweeney, P., and Zadeck, K., "Graph attribution as a specification paradigm," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Boston, MA, November 28-30, 1988), *ACM SIGPLAN Notices* **24**(2) pp. 121-129 (February 1989).

2. Alpern, B., Hoover, R., Rosen, B.K., Sweeney, P.F., and Zadeck, F.K., "Incremental evaluation of computational circuits," pp. 32-42 in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, (San Francisco, CA, Jan. 22-24, 1990), Society for Industrial and Applied Mathematics, Philadelphia, PA (1990).

3. Bricklin, D. and Frankston, B., *VisiCalc Computer Software Program for the Apple II and II Plus*, Personal Software, Inc., Sunnyvale, CA (1979).

4. Dietz, P.F. and Sleator, D.D., "Two algorithms for maintaining order in a list," pp. 365-372 in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, ACM, New York, NY (May 1987).

5. Hoover, R., "Incremental graph evaluation," Ph.D. dissertation and Tech. Rep. 87-836, Dept. of Computer Science, Cornell University, Ithaca, NY (May 1987).

6. Karp, R.M., "On-line algorithms versus off-line algorithms: How much is it worth to know the future?," pp. 416-429 in *Information Processing 92: Proceedings of the IFIP Twelfth World Computer Congress*, ed. J. van Leeuwen, North-Holland, Amsterdam (September 1992).

7. Knuth, D.E., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA (1968, Second Edition: 1973).

8. McGeoch, L.A. and Sleator, D.D. (eds.), *On-Line Algorithms*, American Mathematical Society, Providence, RI (1992).

9. Pardo, R.K. and Landau, R., "Process and apparatus for converting a source program into an object program," U.S. Patent No. 4,398,249, United States Patent Office, Washington, DC (August 9, 1983).

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

11. Ramalingam, G. and Reps, T., "On the complexity of incremental computation," Unpublished report, Computer Sciences Department, University of Wisconsin, Madison, WI (October 1992).

12. Ramalingam, G., “Bounded incremental computation,” Ph.D. dissertation and Tech. Rep. TR-1172, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1993).

13. Reps, T., Teitelbaum, T., and Demers, A., “Incremental context-dependent analysis for language-based editors,” *ACM Trans. Program. Lang. Syst.* **5**(3) pp. 449-477 (July 1983).

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

15. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, NY (1988).

16. Sleator, D.D. and Tarjan, R.E., “Amortized efficiency of list update and paging rules,” *Commun. of the ACM* **28**(2) pp. 202-208 (February 1985).

17. Yeh, D., “On incremental evaluation of ordered attributed grammars,” *BIT* **23** pp. 308-320 (1983).