Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®

Information Processing Letters 88 (2003) 195–200

Information Processing Letters

www.elsevier.com/locate/ipl

Depth-First Discovery Algorithm for incremental topological sorting of directed acyclic graphs

Jianjun Zhou [*], Martin Müller

Department of Computing Science, University of Alberta, Edmonton T6G 2E8, Canada

Received 16 September 2002; received in revised form 19 March 2003

Communicated by F.Y.L. Chin

Abstract

We study the problem of incrementally maintaining a topological sorting in a large DAG. The Discovery Algorithm (DA) of Alpern et al. [Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms, 1990, pp. 32–42] computes a cover $\mathcal{K}$ of nodes such that a solution to the modified problem can be found by changing node priorities within $\mathcal{K}$ only. It achieves a runtime complexity that is polynomially bounded in terms of the minimal cover size k.

The temporary space complexity of DA grows quickly with increasing number of added edges and cover size. We introduce the Depth-First Discovery Algorithm (DFDA), which uses depth-first search to reduce the temporary space of DA from $O(|A| \times \|\mathcal{K}\|)$ to $O(|A| + \|\mathcal{K}\|)$, where $|A|$ is the number of edges to add and $\|\mathcal{K}\|$ is the *extended size* of the cover. DFDA is simpler than DA and performs better in our empirical tests.

Keywords: Topological sorting; Incremental updating; Graph algorithms; DAG; Space complexity

1. Introduction

In many types of computational problems, small changes to a problem instance often have closely related solutions. Computing a solution to a modified problem incrementally from an existing solution can be much more efficient than computing each solution from scratch. Applications of incremental algorithms include program development, interactive systems, text processing, and management and maintenance of consistent information (for example, in spreadsheets) [1,5].

The goal of incremental dynamic priority ordering (topological ordering, topological sorting) is to maintain a correct prioritization of a directed acyclic graph (DAG) as it is modified. A DAG is said to be correctly prioritized if every vertex v in the graph is assigned a scalar priority, denoted by *priority*(v), such that if there is a directed edge from v to w, then *priority*$(v) <$ *priority*(w) [6]. Since deleting edges does not destroy an existing priority order of a DAG, we only discuss edge insertions in this paper. Fig. 1 shows an instance of the incremental updating problem. After adding the edge from the priority 3 node to the priority 2 node, the priorities of the source and destination must be

[*] Corresponding author.
E-mail addresses: jianjun@cs.ualberta.ca (J. Zhou), mmueller@cs.ualberta.ca (M. Müller).

0020-0190/$ – see front matter © 2003 Elsevier B.V. All rights reserved.
doi:10.1016/j.ipl.2003.07.005

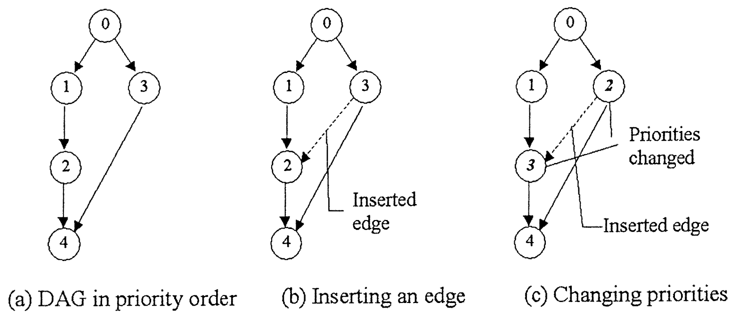

Fig. 1. Inserting an edge and changing priorities.

changed to fix the priority order. In general, changes beyond the endpoints of inserted edges may be necessary. A *cover* is a set of nodes such that the modified problem can be solved by changing only node priorities within the cover.

Definition 1 (*cover [1]*). A set of nodes $\mathcal{K}$ is a cover if, for all nodes v and w in the changed graph,

$$v \xrightarrow{+} w \wedge \mathit{priority}(v) \geqslant \mathit{priority}(w)$$
$$\Longrightarrow v \in \mathcal{K} \vee w \in \mathcal{K},$$

where $v \xrightarrow{+} w$ denotes a nonnull path from v to w.

Alpern et al. argue that the size of $\mathcal{K}$ is not enough to measure the effort of computing the required changes, and propose the *extended size* as a better measure:

Definition 2 (*extended size [1]*). Given a cover $\mathcal{K}$, its extended size

$$\|\mathcal{K}\| \overset{\text{def}}{=} |K| + \left|\text{Touch}(\mathcal{K})\right|,$$

where Touch($\mathcal{K}$) is the set of edges that are incident with nodes in $\mathcal{K}$.

This paper describes an algorithm that computes a cover by a series of searches. The approach is conceptually simple and performs well in practice. We survey previous work and motivate our approach in Section 2, and develop the DFDA algorithm in Section 3. Section 4 presents empirical results, and Section 5 summarizes our contribution.

2. Previous work

We discuss three algorithms for the problem of incrementally updating a topological ordering. Hoover's method [2] inserts one edge at a time. However, in typical applications edges are added in larger batches [1]. Based on Hoover's work, Alpern et al. gave an algorithm for simultaneously inserting multiple edges. This algorithm was the first with a runtime complexity that is polynomially bounded in the general case in terms of the minimal extended size of the cover [1]. However, when dealing with large DAGs with a large number of edges to insert, their algorithm requires a lot of working memory. Our work focuses on resolving this problem. Ramalingam et al. [5] developed a method for the more general problem of managing incremental changes in constraint systems, which can be used to maintain a topological sorting. However, the runtime of this algorithm is not polynomially bounded, and it can produce covers that are much larger than the minimal one.

Our work is based on the two-step algorithm of Alpern et al.: first the Discovery Algorithm (DA) marks a cover, then the Reassignment Algorithm (RA) assigns consistent priorities to nodes in the cover. DA processes all added edges simultaneously, needs $2|A|$ different priority queues, and requires $\Theta(|A|)$ storage per visited node in the worst case. The purpose of marking nodes for all edges simultaneously is to avoid repeatedly reassigning the priorities of nodes, which can happen if Hoover's single-edge algorithm is called for each edge.

While DA uses a breadth-first approach to organize the processing of different inserted edges, our new algorithm DFDA works depth-first. Added edges are processed one by one, extending the cover, while reducing the $O(|A| \times \|\mathcal{K}\|)$ temporary storage requirements of DA to $O(|A| + \|\mathcal{K}\|)$. The improved memory efficiency also enhances the overall processing speed in practice. Both algorithms achieve the same theoretical runtime complexity of $O(|A| k \log k)$, where k is the minimum extended size for a cover.

Given a correctly prioritized DAG and a cover $\mathcal{K}$, the Reassignment Algorithm (RA) checks that the graph is cycle-free and reassigns priorities to the nodes in $\mathcal{K}$ with a worst case runtime of $O(\|\mathcal{K}\| + |\mathcal{K}| \log |\mathcal{K}|)$. In practice, RA is faster than DA and uses less memory, so improving DA enhances the performance of the whole system.

3. Depth-First Discovery Algorithm (DFDA)

Our Depth-First Discovery Algorithm (DFDA) is based on the Discovery Algorithm (DA) [1]. Both DA and DFDA mark the nodes of a cover $\mathcal{K}$ by considering pairs (x, y) of unmarked nodes such that $x \xrightarrow{+} y$. If the priorities are out of order, then at least one of the nodes is marked. The algorithms use a counter called the *edge value* (*ev*) in each visited node. *ev* is initialized to the degree of the node, and is decremented during algorithm execution as described below.

Initially, the DAG is assumed to be prioritized correctly. After adding a batch of edges, each new edge e searches backward from its source and forward from its destination, forming two frontiers which are stored in priority queues: a *backwardFrontier* that keeps the largest priority, and a *forwardFrontier* that keeps the smallest priority on top.

In DA, there is one pair of priority queues for each new edge e. The nodes b and f at the top of the two queues of e represent the set of all pairs (x, y) such that $x \xrightarrow{+} y$ along a path that includes b, e and f. If the priorities of b and f are in order, then so are the priorities of all x and y. Otherwise, b or f (or both) are marked as visited by e, and search continues until either a frontier becomes empty or a correctly ordered pair (b, f) is found.

The main difference between DA and DFDA is the handling of inserted edges. DA expands search frontiers of inserted edges one step at a time, and needs two visited fields for each (visited node, inserted edge) pair. DFDA keeps pushing the frontiers of one inserted edge until the frontier node pairs are in order, or until at least one frontier becomes empty. The visited fields are reset before processing the next inserted edge. DFDA only needs two fields for each visited node.

There are two kinds of dependencies between computations in both algorithms. One is for a single inserted edge. The first time a node x is visited by an edge e while searching forward, x is marked as visited by e. When the search from e meets x again along a different path, it is skipped, and e continues searching the successors of x. Another dependency is between different inserted edges. If node x is already marked by e, then all other edges e' will skip x and search beyond it.

An important feature of both DA and DFDA is how they choose which nodes to mark as part of the cover. For every pair (b, f), the node with smaller *ev* is marked, and the *ev* counters of both nodes are decreased by this smaller value. In case of equal *ev* values, both nodes are marked. This is the key technique to achieve bounded runtime complexity.

The process is illustrated in Fig. 3. For the case of DA, after adding e and e', x and y are the frontiers of edge e, and x and z the frontiers of e'. Assuming the algorithm deals with e first, x is marked since $x.ev < y.ev$, and is extended. The *ev* of both x and y is reduced by $x.ev = 3$. For edge e', node x is skipped since it was already marked by e. Next, DA checks the pairs (w, y) and (w, z).

They are in order, so the so DA stops. For DFDA, first (x, y) and (w, y), then (x, z) and (w, z) are checked.

Let A be the set of added edges. DA uses $\Theta(|A|)$ worst case space for marking in each visited node. Temporary space is bounded by $O(|A| \times \|\mathcal{K}\|)$, since only nodes in $\mathcal{K}$, and neighbor nodes will be visited. Let k be the minimal possible value of $\|\mathcal{K}\|$. Then DA computes a cover with extended size $\|\mathcal{K}\| \leqslant 3k$ in a worst case time of $O(|A| k \log k)$ [1].

Pseudocode for DFDA is shown in Fig. 2. *ProcessEdge* is called for each inserted edge. The source and destination of an edge e are denoted by *e.source* and *e.destination*. The edge value of a node n is indicated by *n.ev*. Procedures *GetBackward* and *GrowBackward* are analogous to the procedures *GetForward* and *GrowForward* shown.

```
Main()
  for all inserted edges $e$
    $e.destination.visitedForward := true$;
    forwardFrontier.Clear(); forwardFrontier.evPush($e.destination$);
    $e.source.visitedBackward := true$;
    backwardFrontier.Clear(); backwardFrontier.evPush($e.source$);
    ProcessEdge();
    Reset all visitedForward and visitedBackward fields;

PriorityQueue.evPush($x$)
  $x.ev :=$ degree$(x)$;
  Push($x$);

ProcessEdge()
  while $f :=$ GetForward() $\wedge$ $b :=$ GetBackward() $\wedge$ $b.priority \geqslant f.priority$
    $\mu := \min(f.ev, b.ev)$;
    if $f.ev = \mu$
      forwardFrontier.Pop();
      $f.marked := true$;
      GrowForward$(f)$;
    if $b.ev = \mu$,
      backwardFrontier.Pop();
      $b.marked := true$;
      GrowBackward$(b)$;
    $f.ev := f.ev - \mu$; $b.ev := b.ev - \mu$;

GetForward()
  do
    if forwardFrontier.IsEmpty() return $Null$;
    $x :=$ forwardFrontier.Top();
    if $x.marked$
      forwardFrontier.Pop();
      GrowForward($x$);
  while $x.marked$;
  return $x$;

GrowForward($x$)
  for all $y$ with $x \to y$ do
    if $\sim y.visitedForward$
      $y.visitedForward := true$;
      if $y.marked$ GrowForward($y$);
      else forwardFrontier.evPush($y$);
```

Fig. 2. The Depth-First Discovery Algorithm.

Theorem 1. *DFDA needs temporary space of size* $O(|A| + \|\mathcal{K}\|)$.

Proof. Besides storing the input DAG, the $|A|$ added edges and two frontiers of size $O(\|\mathcal{K}\|)$, the algorithm stores an edge value and only two visited values (visitedForward and visitedBackward) for each of at most $\|\mathcal{K}\|$ visited nodes. □

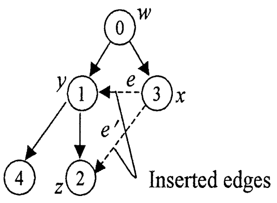

Fig. 3. Communication between edges.

Theorem 2. *Given a correctly prioritized directed graph and a set A of newly added edges, DFDA marks a set of nodes $\mathcal{K}$ such that*

(a) *$\mathcal{K}$ is a cover;*
(b) *$\|\mathcal{K}\| \leqslant 3k$, where k is the minimum extended size for a cover;*
(c) *the worst case running time is* $O(|A|k\log k)$.

Proof (*Sketch*). (a) Any nonnull path $P: w \xrightarrow{+} z$ with Priority$(w) \geqslant$ Priority(z) must contain inserted edges. As in DA, the search for an inserted edge skips any marked node as the frontier moves along P. A proof by induction over the path length as in Theorem 2 of [1] shows that w or z (or both) will be marked.

(b) Because $\mathcal{K}$, is a cover, and like DA, DFDA uses the smaller edge value for selecting nodes to mark, by the same argument as in Theorem 2 of [1], it follows that $\|\mathcal{K}\| \leqslant 3k$.

(c) Insertion and deletion in a priority queue with $O(k)$ elements can be performed in time $O(\log k)$. Each node in $\mathcal{K}$ and the neighbor nodes of $\mathcal{K}$ will be added to a queue at most once for each e, and each time the algorithm checks the order of two nodes, at least one node is removed from the queue. The visited values must be reset in at most $\|\mathcal{K}\|$ nodes for each e. The overall time complexity is $O(|A|k\log k + |A|k) = O(|A|k\log k)$. $\square$

4. Empirical results

We compared the performance of DA and DFDA on random DAGs generated by DagAlea [4]. For uniformity, algorithms were implemented using LEDA [3], with Fibonacci heap priority queues. All experiments were performed on a Pentium III 700 MHz workstation with 768 MB of memory.

Fig. 4 compares the runtime on 1000 random DAGs with 1000 vertices and 3000 edges each. A small, randomly selected set of edges was removed from each DAG, then added as a batch. We repeated this process 10 times with different random batches for each DAG, so each point in the plot corresponds to an average over 10×1000 runs. With increasing batch size, DFDA benefits from its smaller temporary storage and simpler algorithm.

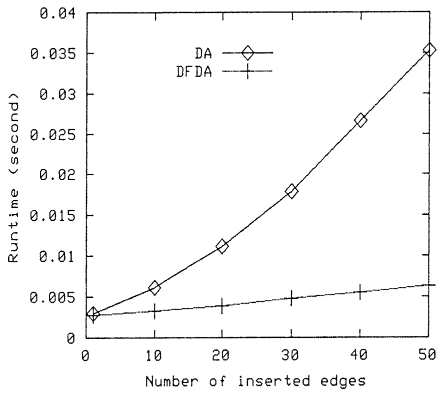

Fig. 4. Comparison of runtime as batch size varies.

We also compared cover size and extended cover size for DA and DFDA experimentally. The observed differences in sizes were very small.

5. Summary

We proposed DFDA as an improvement to the Discovery Algorithm of Alpern et al. that reduces the temporary space from $O(|A| \times \|\mathcal{K}\|)$ to $O(|A| + \|\mathcal{K}\|)$, where $|A|$ is the number of added edges and $\|\mathcal{K}\|$ the extended size of the cover. DFDA achieves much better runtime performance. Processing inserted edges by a series of searches, not simultaneously, also simplifies the algorithm.

Acknowledgements

We are grateful to Lorna Stewart for insightful discussions and feedback on our manuscript, and to the anonymous reviewers for comments that greatly helped to improve the presentation of this paper.

References

[1] B. Alpern, R. Hoover, B. Rosen, P. Sweeney, F.D. Zadeck, Incremental evaluation of computational circuits, in: Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms, 1990, pp. 32–42.
[2] R. Hoover, Incremental graph evaluation, Technical Report 87-836 (PhD Thesis), Dept. of Computer Science, Cornell University, Ithaca, NY, May 1987.

[3] K. Mehlhorn, S. Näher, LEDA: a platform for combinatorial and geometric computing, Comm. ACM 38 (1995) 96–102.

[4] G. Melancon, I. Herman, DAG drawing from an information visualization perspective, in: Proc. of the Joint Eurographics and IEEE TCVG Symposium on Visualization, 2000; also available online: http://www.cwi.nl/InfoVisu.

[5] G. Ramalingam, J. Song, L. Joskowicz, R.E. Miller, Solving systems of difference constraints incrementally, Algorithmica 23 (1999) 261–275.

[6] G. Ramalingam, T. Reps, On competitive on-line algorithms for the dynamic priority-ordering problem, Inform. Process. Lett. 51 (3) (1994) 155–161.