Theoretical Computer Science 48 (1986) 273-281
North-Holland



# AMORTIZED EFFICIENCY OF A PATH RETRIEVAL DATA STRUCTURE*

G.F. ITALIANO

*Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", I-00184 Roma, Italy*

Communicated by M. Nivat
Received June 1986

**Abstract.** A data structure is presented for the problem of maintaining a digraph under an arbitrary sequence of two kinds of operations: an ADD operation that inserts an arc in the digraph, and a SEARCHPATH operation that checks the presence of a path between a pair of nodes. Our data structure supports both operations in $O(n)$ amortized time and requires $O(n^2)$ space, where $n$ is the number of nodes in the digraph.

## 1. Introduction

There are some recent results about dynamization of well known graph problems such as connectivity by Even and Shiloach [3], transitive closure by Ibaraki and Katoh [7], minimum spanning trees by Frederickson [4] and shortest path by Rohnert [9].

Our problem (which we call the *path-insert problem*) can be formalized as follows: Perform a sequence of intermixed operations of the following two kinds on directed graphs:

ADD$(i, j)$: insert one arc between nodes $i$ and $j$ in the graph;
SEARCHPATH$(i, j)$: check a path from node $i$ to node $j$; if such a path exists, return it, arbitrarily choosing one when several paths are permitted.

For undirected graphs this problem can be solved using the well known set-union algorithms [12] in $O(n + m\alpha(m + n, n))$ time, where $n$ is the total number of nodes, $m$ the total number of operations, and $\alpha(x, y)$ a very slowly growing function. In the case of directed graphs the problem seems to be much more time consuming.

In fact, if we store only the arcs pointed out by the user during the add operations, every arc insertion can be accomplished in a constant time while path checking can require $O(n^2)$ worst-case time. On the other side, if complete information about paths (e.g., the transitive closure of the graph [2]) is maintained, paths can be

* This research was partially supported by project MPI "Progetto e Analisi di Algoritmi" and by Selenia S.p.A.

retrieved in $O(n)$ time, while every new arc insertion can involve so much updates as to require $O(n^2)$ time.

Furthermore, even a dynamic maintenance of the transitive closure does not introduce any improvement to these bounds since the algorithm proposed in [7], though optimal for path checking, can easily be shown to take $O(n^3)$ time in processing a particular sequence of $O(n)$ add operations, thus requiring at least $O(n^2)$ amortized time.

The trade-off between add and searchpath operations was firstly investigated in [8], where a data structure requiring $O(n \log n)$ amortized time for both operations was proposed.

We now present a new data structure that supports the above operations in $O(n)$ amortized time and requires $O(n^2)$ space.

The remainder of this paper consists of four sections. In Section 2 graph terminology will be introduced. A description of the data structure is given in Section 3, while in Section 4 we analyse its amortized efficiency. Finally, Section 5 contains some concluding remarks.

## 2. Graph definitions

We assume that the reader is familiar with the standard graph terminology as contained in [6].

In particular, a *directed graph* $G=(V, E)$ (sometimes called a *digraph*) is a finite set $V=\{1, 2, \ldots, n\}$ of *nodes* and a finite set $E$ of *arcs* such that each arc $e$ has a *head* $h(e) \in V$ and a *tail* $t(e) \in V$. We consider the arc $e$ as leading from $h(e)$ to $t(e)$ and we say that the arc $e$ leaves $h(e)$ and enters $t(e)$.

A *path* $p=e_1, e_2, \ldots, e_k$ is a sequence of arcs such that $t(e_i)=h(e_{i+1})$ for $1 \leq i \leq k-1$. The path is from $h(p)=h(e_1)$ to $t(p)=t(e_k)$ and contains arcs $e_1, e_2, \ldots, e_k$ and nodes $h(e_1), h(e_2), \ldots, h(e_k), t(e_k)$. The length of a path is the number of arcs it contains. As a special case, a single node denotes a path of length 0 from itself to itself. A *cycle* is a nonempty path from a node to itself.

A node $v$ is *reachable* from a node $u$ if there is a path from $u$ to $v$: in such a case $u$ is said to be an *ancestor* of $v$ and $v$ a *descendant* of $u$. If in addition $u \neq v$, $u$ is a *proper ancestor* of $v$ and $v$ is a *proper descendant* of $u$. If there is an arc from $u$ to $v$, then $v$ is *adjacent* to $u$.

If $G=(V, E)$ is a digraph, the digraph which has the same vertex set as $G$ but has an arc from $u$ to $v$ if and only if there is a path from $u$ to $v$ in $G$, is called the *transitive closure* of $G$.

A digraph with no cycles is called a *directed acyclic graph* (dag). A *rooted tree* is a dag satisfying the following properties:

(1) there is only one node, called the *root*, which no arcs enter;
(2) every node except the root has exactly one entering arc;
(3) there is a path (which is unique) from the root to each node.

Given a digraph $G=(V, E)$, a *spanning tree* is a rooted tree $S=(V, T)$ such that $T \subseteq E$.

## 3. The data structure

In this section we present a data structure to maintain a digraph $G=(V, E)$ under an arbitrary sequence of path-insert operations.

The basic idea is to represent the transitive closure of $G$. That is, we associate to each node $x \in V$ a set desc($x$), which contains all the descendants of $x$. In order to easily extract information about paths, the sets desc($x$), $\forall x \in V$, are organized as spanning trees.

In addition, while updating the spanning tree structures during new arc insertions, we shall perceive the need of accessing each node in the spanning trees very quickly. To achieve this goal we make use of an $n \times n$ matrix of pointers, defined as follows:

$$\text{index}(i,j)\begin{cases}\text{points to the node } j \text{ in desc}(i) & \text{if } j \in \text{desc}(i),\\ \text{contains a } \textbf{null} \text{ pointer} & \text{otherwise.}\end{cases}$$

Figure 1 shows how the presence of a path from $i$ to $j$ can be checked by simply examining the entry index($i, j$). In fact, if index($i, j$) contains a **null** pointer, then there is no path from $i$ to $j$. Otherwise, index($i, j$) allows to easily locate the position of $j$ in the spanning tree of the descendants of $i$.

From this point of view, the matrix index generalizes the adjacency matrix of the transitive closure of $G$.

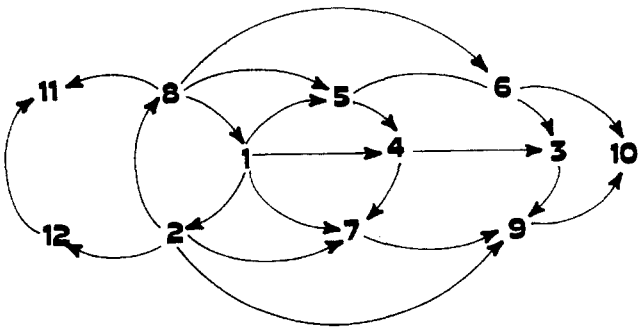
*The algorithm* PATH-INSERT

The data structure is initialized as follows:

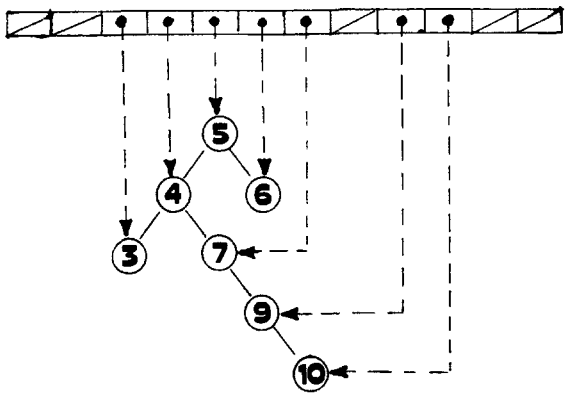
```
procedure INITIALIZE;
begin
  for i := 1 to n do
  begin
    desc(i) := "empty tree";
    for j := 1 to n do index(i, j) := null
  end
end (* initialize *);
```

Clearly, procedure INITIALIZE requires $O(n^2)$ time. However, this preprocessing time can be reduced to $O(n)$ by initializing each entry of the matrix index the first time it is accessed (see, for instance, [2, p 71]).

The presence of a path between nodes $i$ and $j$ can be checked by simply traversing the spanning tree desc($i$). If reversal pointers to the parent for each node in the spanning trees are maintained, a bottom-up traversal from $j$ to $i$ in desc($i$) takes at most $O(k)$ time units to return a path from $i$ to $j$, where $k \leq n$ is the length of the




(a)




(b)

Fig. 1. (a) A digraph $G$. (b) The spanning tree desc(5) together with the 5th row of the matrix index.

achieved path:

```
procedure SEARCHPATH(i, j, T);      (* T is the achieved path *)
begin                               (* from i to j *)
  T := ∅;
  if index(i, j) ≠ null then        (* j is reachable from i *)
  begin
    p := index(i, j);               (* locate j in desc(i) *)
    T := {j};
    repeat                          (* go up in desc(i) ... *)
      p := parent(p);
      T := key(p) ∘ T
    until parent(p) = null          (* ... until the root i *)
  end                               (* is reached *)
end (* SEARCHPATH *);
```

On the other hand, new arc insertions can be carried out as the procedure ADD shows:

```
procedure ADD(i, j);          (* insert an arc from i to j *)
begin
  if index(i, j) = null then  (* there was no previous path from i to j *)
    for x := 1 to n do
      if (index(x, i) ≠ null) and (index(x, j) = null) then
      (* the arc (i, j) gives rise to a new path from x to j *)
        MELD(x, j, i, j)      (* update desc(x) by means of desc(j) *)
end (* ADD *);
```

Note that an arc between nodes $i$ and $j$ is inserted only if there was no previous path from $i$ to $j$. In fact, if such a path already exists, the arc $(i, j)$ does not add any further information to our problem.

Furthermore, the insertion of the arc $(i, j)$ could induce a new connection between any ancestor $x$ of $i$ and the descendants of $j$ only if there was no previous path from $x$ to $j$, i.e., only if index$(x, i) \neq$ **null** and index$(x, j) =$ **null**. In such a case, the spanning tree desc$(x)$ must be updated, taking into account the descendants of $j$. This can be accomplished by melding the two spanning trees in $x$ and $j$, namely by

(i) pruning a copy of desc$(j)$, i.e., by eliminating from it the nodes already in desc$(x)$;

(ii) linking this pruned copy to the node $i$ in desc$(x)$ and by updating in a proper way the $x$th row of the matrix index.

Obviously, the melding process does not change desc$(j)$ (see Fig. 2).

Procedure MELD$(x, j, u, v)$ merges desc$(x)$ and the subtree of desc$(j)$ rooted at $v$; $u$ is the node of desc$(x)$ to which the pruned subtree will be linked:

```
procedure MELD(x, j, u, v);
begin
  "create a new node v pointed by index (x, v)";
  "insert it in desc(x) as a child of u";
  for each w child of v in desc(j) do
    if index(x, w) = null then MELD(x, j, v, w)
end (* MELD *):
```

Clearly, if the spanning tree desc$(x)$ must be updated during an ADD$(i, j)$ operation, it is sufficient to perform a MELD$(x, j, i, j)$, while in the subsequent recursive calls, $u$ will be the parent of $v$ in desc$(j)$.

Furthermore, if there is a node $v$ in desc$(j)$ such that index$(x, v) \neq$ **null** (i.e., $v$ is in desc$(x)$), then all the descendants of $v$ already belong to desc$(x)$. This gives reason for the fact that a MELD$(x, j, u, v)$ is called only if index$(x, v) =$ **null**. As a straightforward consequence, the melding process cannot traverse arcs in desc$(j)$ whose endpoints are both in desc$(x)$: this will be a basic point in the analysis developed in the following section.

The correctness of the whole algorithm is almost obvious since it maintains the transitive closure of the digraph while new arcs are inserted [8].

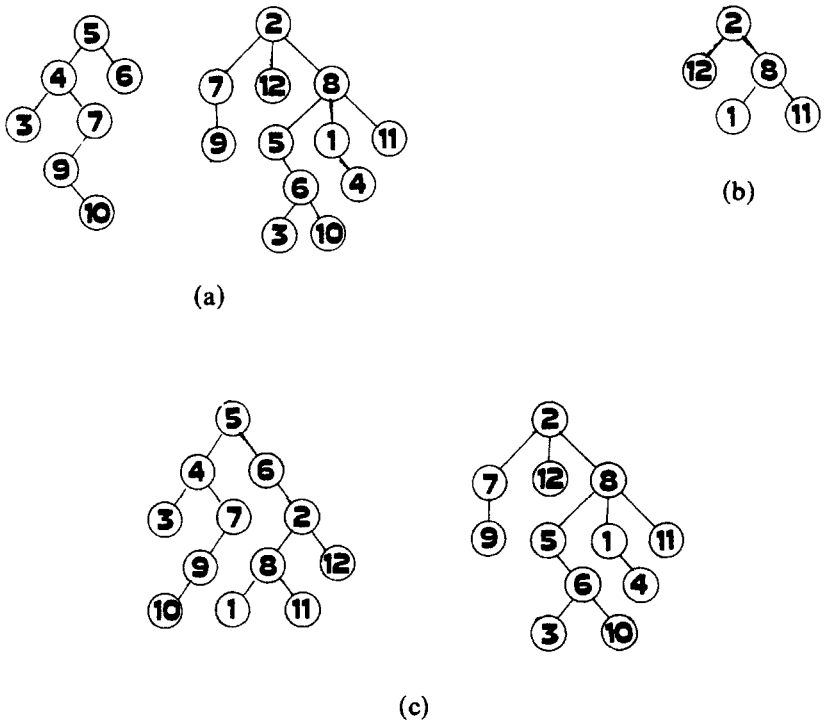



Fig. 2. Updating desc(5) by means of desc(2) caused by an ADD(6, 2) operation in the digraph of Fig. 1(a). (a) desc(5) and desc(2) before the ADD(6, 2) operation. (b) The pruned copy of desc(2). (c) desc(5) and desc(2) after the ADD(6, 2) operation.

## 4. Analysis of the algorithm

Our data structure is efficient in an amortized sense: any particular operation may be slow but any sequence of operations must be fast. An amortized bound is often more useful than average-case or worst-case, per operation bounds, as pointed out by Sleator and Tarjan in [10].

We analyse the amortized running time of the path-insert operations by using the 'potential' technique of Tarjan [5, 11]. We assign to each node a real number $\varphi$ called the *potential* of the node and define the *amortized time a* of a path-insert operation to be its actual running time $t$ plus the increase it causes in potential (a decrease in potential counts negatively):

$$a = t + \sum_{v \in V} \varphi'(v) - \sum_{v \in V} \varphi(v).$$

With this definition, the actual time of a sequence of $m$ operations is equal to the total amortized time plus the decrease in potential over the entire sequence:

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \sum_{v \in V} \varphi(v) - \sum_{v \in V} \varphi'(v).$$

Thus the total running time can be estimated by choosing a potential function and bounding $\varphi(v)$, $\varphi'(v)$, and $a_i$ for each $i$.

To apply this technique, we need a new definition.

**Definition 4.1.** Given any node $x$ in the digraph $G=(V, E)$, the following set

$$\text{vis}(x)=\{(u, v)\in E \mid u \text{ is a descendant of } x\}$$

is called the *visibility* of $x$.

**Remark 4.2.** vis($x$) contains only the arcs whose endpoints are both descendants of $x$. Hence, as we have previously specified, all the arcs in vis($x$) cannot be examined while updating desc($x$).

Referring to |desc($x$)| as the number of nodes it contains, we define the potential of a node $x$ as

$$\varphi(x)=-(|\text{vis}(x)|+3|\text{desc}(x)|).$$

Furthermore, the potential of the data structure is

$$\Phi=\sum_{x\in V}\varphi(x).$$

The amortized complexity of the add operation can now be analysed as the following argument shows. A MELD($x, j, i, j$) examines $h_1$ arcs in desc($j$) and adds $h_2$ arcs to desc($x$), where $h_2\leq h_1+1\leq n$. Only the potential in $x$ is modified, with a net decrease of $(h_1+3h_2)$ since:

(i) Remark 4.2 guarantees that, while updating desc($x$), we cannot examine arcs already belonging to vis($x$);

(ii) each examined arc will be in the new visibility of $x$;

(iii) the size of desc($x$) will increase exactly by $h_2$.

While inserting a node $v$ in a spanning tree desc($x$) as a child of $u$, the following three linking steps are required:

- make $v$ a child of $u$ in desc($x$);
- set to $u$ the reversal pointer of $v$ (parent($v$)) in desc($x$);
- let index($x, v$) point to the node $v$ in desc($x$).

Thus, if we charge one unit of time for each arc-traversing and -linking step, updating a spanning tree has an amortized time of $O(h_1+3h_2-h_1-3h_2)\sim O(1)$.

Hence, the amortized time of an ADD($i, j$) operation is $O(n)$ since it requires the updates of at most $n$ spanning trees.

The searchpath operation has an amortized time of $O(k)$, where $k$ is the length of the achieved path: in fact, a bottom-up traversal of a spanning tree requires $O(k)$ actual time and changes no potential.

As far as the storage utilization of the data structure is concerned, our implementation uses $n$ trees of size at most $n$ (desc($x$), $\forall x\in V$), and one $n\times n$ matrix of pointers (index($i, j$)). Thus, the space complexity is $O(n^2)$.

This leads to the following theorem.

**Theorem 4.3.** *The data structure supports both* ADD *and* SEARCHPATH *operations in* $O(n)$ *amortized time and requires* $O(n^2)$ *space, where n is the number of nodes in the digraph.*

## 5. Open problems and future work

In this paper we have described fast algorithms for the path-insert problem on digraphs by presenting a data structure that supports both ADD and SEARCHPATH operations in $O(n)$ amortized time and requires $O(n^2)$ space. This result may be generalizable in various directions.

First of all, the question of whether there is an algorithm that is efficient in a worst-case, per operation sense remains open.

Furthermore, this problem may be extended to graphs with arc lengths, namely searching the shortest path between pairs of nodes while inserting new arcs. In contrast to the unweighted case, where a path from $x$ to $y$ once established never needs to be updated again, in this case, a newly added arc $(u, v)$ may create a path shorter than an existing old one from $x$ to $y$. Hence, the computational saving implied by our data structure seems to be no longer valid. In any case, is it possible to achieve better than $O(n^2)$ bounds?

Finally, a more general path-insert problem may also be defined on hypergraphs. This generalization appears useful in many applications [1] as well as being of theoretical interest.

## Acknowledgment

I am very grateful to Giorgio Ausiello and Alberto Marchetti Spaccamela for their encouragement and their helpful comments.

## References

[1] G. Ausiello, A. D'Atri and D. Saccà, Minimal representation of directed hypergraphs, *SIAM J. Comput.*, to appear.

[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[3] S. Even and Y. Shiloach, An on-line edge deletion problem, *J. ACM* **28** (1981) 1–4.

[4] G.N. Frederickson, Data structures for on-line updating of minimum spanning trees, *Proc. 15th ACM Symp. on Theory of Computing* (1983) 252–257.

[5] M.L. Fredman and R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Proc. 25th IEEE Symp. on Foundations of Computer Science* (1984) 338–346.

[6] F. Harary, *Graph Theory* (Addison-Wesley, Reading, MA, 1972).

[7] T. Ibaraki and N. Katoh, On-line computation of transitive closure of graphs, *Inform. Process. Lett.* **16** (1983) 95–97.

[8] G.F. Italiano, Transitive closure for dynamic graphs, Tech. Rept. 01.86, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", 1986.

[9] H. Rohnert, A dynamization of the all pairs least cost path problem, *Proc. 2nd Symp. on Theoretical Aspects of Computer Science* (1985) 279-286.

[10] D.D. Sleator and R.E. Tarjan, Amortized efficiency of list update rules, *Proc. 16th ACM Symp. on Theory of Computing* (1984) 488-492.

[11] R.E. Tarjan, Amortized computational complexity, *SIAM J. Algebraic Discrete Methods* **6** (1985) 306-318.

[12] R.E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *J. ACM* **31** (1984) 245-281.