Dynamic Data Structures for Series Parallel Graphs

Giuseppe F. Italiano
Alberto Marchetti Spaccamela
Umberto Nanni

CUCS-417-89

Dynamic Data Structures for Series Parallel Digraphs *

Giuseppe F. Italiano †
Department of Computer Science
Columbia University, New York, NY 10027
and
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Roma, Italy

Alberto Marchetti Spaccamela
Dipartimento di Matematica Pura e Applicata
Università di L'Aquila, L'Aquila, Italy

Umberto Nanni ‡
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Roma, Italy

January 1989

Abstract

We consider the problem of dynamically maintaining general series parallel directed acyclic graphs (GSP dags), two-terminal series parallel directed acyclic graphs (TTSP dags) and looped series parallel directed graphs (looped SP digraphs). We present data structures for updating (by both inserting and deleting either a group of edges or vertices) GSP dags, TTSP dags and looped SP digraphs of m edges and n vertices in $O(\log n)$ worst-case time. The time required to check whether there is a path between two given vertices is $O(\log n)$, while a path of length k can be traced out in $O(k + \log n)$ time. For GSP and TTSP dags, our data structures are able to report a regular expression describing all the paths between two vertices x and y in $O(h + \log n)$, where $h \leq n$ is the total number of vertices which are contained in paths from x to y. Although GSP dags can have as many as $O(n^2)$ edges, we use an implicit representation which requires only $O(n)$ space. Motivations for studying dynamic graphs arise in several areas, such as communication networks, incremental compilation environments and the design of very high level languages, while the dynamic maintenance of series parallel graphs is also relevant in reducible flow diagrams.

*Work partially supported by the Italian MPI National Project "Algoritmi e Strutture di Calcolo".
†Partially supported by NSF Grants DCR-85-11713, CCR-86-05353 and by an IBM Graduate Fellowship.
‡Partially supported by Selenia S.p.A.

1 Introduction

Significant progress has been recently made in the design of algorithms and data structures for dynamic graphs. These data structures support insertions and deletions of edges and/or vertices in a graph, in addition to several types of queries. In the following, we will restrict our attention to insertions/deletions of edges. The same algorithms given in this case can easily be generalized to take into account insertions/deletions of vertices by means of standard techniques.

In particular, much attention has been devoted to the on-line computation of the connected components of graphs [6, 7, 13, 14, 16, 17, 18, 20, 23, 24, 26, 28, 34]. The problem consists of maintaining an underlying graph under an intermixed sequence of operations of the following kind.

- *add*(x, y) : insert an edge between vertices x and y.
- *delete*(x, y) : remove the edge between vertices x and y.
- *query*(x, y) : return *true* if there is a path from x to y; return *false* otherwise.
- *report*(x, y) : return an arbitrarily chosen path from x to y, if one exists.

We will refer to this problem also as the dynamic maintenance of the transitive closure of a graph. In the remainder of this paper, we denote by m the number of edges and by n the number of vertices in a graph.

Motivations for studying dynamic data structures for graph problems arise in several areas including, among others, communication networks, incremental compilation environments [9, 15] and the design of very high level languages for incremental computations [35]. So far, the following three dynamic problems have been considered:

(P1) (*Insertion Problem*) Perform an arbitrary sequence of add, query and report operations.

(P2) (*Deletion Problem*) Perform an arbitrary sequence of delete, query and report operations.

(P3) (*Fully Dynamic Problem*) Perform an arbitrary sequence of add, delete, query and report operations.

Depending on the different kind of graphs, these problems can be solved in the times given below. For undirected graphs, by using the set union data structures of Tarjan [31], *(P1)* can be solved quite efficiently. Indeed, each add can be supported in $O(1)$ time, while query and report can be carried out respectively in $O(\alpha(q, n))$ and $O(k + \alpha(q, n))$ amortized time [30], where q is the total number of query and report operations, k is the length of the achieved path and α is a very slowly growing function, a functional inverse of Ackermann's function [31]. Frederickson [14] proposed a data structure for solving *(P3)* (and therefore also *(P2)*), in which each insertion or deletion of edges can be performed in $O(\sqrt{m})$ worst-case time, while still allowing $O(1)$ queries about the transitive closure. The data structure can be easily modified in order to trace out paths in linear time. Frederickson's data structure takes advantage of the topological properties of the underlying graph. When the graph is planar, things can be done more efficiently and the update bound becomes $O(\log^2 n)$ in the worst case, provided that the updates leave the graph planar.

As for directed graphs (hereafter referred to also as *digraphs*), the problem of computing on line the transitive closure was first tackled by Ibaraki and Katoh [16]. They proposed a data structure which requires a total of $O(n^3)$ time to solve *(P1)* and $O(n^2(m + n))$ time to solve *(P2)*. The overall time needed for solving *(P1)* was later reduced to $O(mn)$ [17]. Also *(P2)* can be solved in a

total of $O(mn)$ time if the original digraph is acyclic [18], thus improving one order of magnitude on the cyclic case. The data structures in [17, 18] are able to answer each connectivity question in $O(1)$ time and to report a path of length k in $O(k)$ time. Their space complexity is $O(n^2)$. Similar bounds were recently achieved by La Poutré and van Leeuwen [20] and by Yellin [34].

In a first comparison to their undirected counterparts, we notice that the time required to solve these problems increases by almost one order of magnitude. For *(P1)*, the update amortized time bound becomes $O(n)$ instead of $O(\alpha(q, n))$. There is also a loss of one order of magnitude for *(P2)*. Even worse, no efficient fully dynamic data structure is known for general digraphs. This gives empirical evidence that reachability for directed graphs is "harder" than for undirected graphs, an assertion which has been recently supported by means of theoretical arguments [5].

This suggests the investigation of classes of digraphs for which the best bounds known for the general case can be improved. Indeed Preparata and Tamassia [23, 28] designed a fully dynamic data structure for planar *st*-graphs, a subclass of planar acyclic digraphs with exactly one source and one sink, both on the external face. Their data structure is able to perform each add, delete and query operation in $O(\log n)$ time and reports a path of length k in $O(k + \log n)$ time, provided that the digraph remains *st*-planar after each update. Since planar digraphs have at most $O(n)$ edges, their data structure requires $O(n)$ space.

In this paper, we further investigate classes of digraphs for which the best bounds known for general digraphs can be improved. In particular, we consider three classes of series parallel digraphs: general series parallel directed acyclic graphs (in short *GSP dags*) [32], two-terminal series parallel directed acyclic graphs (in short *TTSP dags*) [32] and looped series parallel directed graphs (in short *looped SP digraphs*) [1]. We show how to efficiently maintain these classes of digraphs under a sequence of update operations, each of which leaves the digraph series parallel. In all the three cases, the time required for each update (both insertions and deletions of either a group of edges or one vertex) is $O(\log n)$, while the presence of a path can be checked in $O(\log n)$. A path of length k can be traced in $O(k + \log n)$. For GSP and TTSP dags, a regular expression describing all the paths between two vertices x and y can be reported in $O(h + \log n)$ time, where $h \leq n$ is the total number of vertices in paths from x to y. The space required by all the data structures is $O(n)$. All these bounds improve the best bounds previously known for general digraphs.

Series parallel digraphs and their undirected counterparts arise naturally in many applications such as network design [10] and scheduling under constraints [21, 22]. In [8, 19, 27] linear time algorithms are given for solving many problems on such graphs, including problems which are NP-complete for the general case. Dynamic maintenance of series parallel digraphs arises in different areas such as communication networks and reducibility of flow diagrams [2].

Our approach seems to be appealing for several reasons. First, despite the fact that GSP dags can have as many as $O(n^2)$ edges, we use an implicit representation which reduces to $O(n)$ the space complexity. Second, even if each single update operation operation can either insert or delete as many as $O(n^2)$ edges, each of these operations can still be supported in $O(\log n)$ worst-case time. Finally, this is the first case that we know of, for which an efficient fully dynamic data structure maintains on line the transitive closure of a non-trivial class of digraphs which contains cyclic digraphs (namely, the class of looped SP digraphs).

The remainder of the paper consists of five sections. In section 2 we give some preliminary definitions. Data structures for GSP and TTSP dags are presented in section 3 and their time and space complexity analyzed in section 4. Looped SP digraphs are considered in section 5. Section 6 lists some open problems and concluding remarks.

2 Preliminary Definitions

We assume that the reader is familiar with the standard graph theoretical terminology as contained for instance in [4]. In addition, we recall that, given a digraph G, a *source* is a vertex with no entering edges and a *sink* is a vertex with no leaving edges. We now introduce some terminology about series parallel digraphs. A *Two Terminal Series Parallel digraph* [32] (in short *TTSP digraph*) with terminals (source and sink) s and t can be produced by a sequence of the following operations.

- Create a new digraph consisting of the edge (s, t).
- Given two TTSP digraphs G_1 and G_2 with terminals s_1, t_1, s_2, and t_2, form a new digraph G_p by identifying $s = s_1 = s_2$ and $t = t_1 = t_2$ *(parallel composition).*
- Given two TTSP digraphs G_1 and G_2 with terminals s_1, t_1, s_2, and t_2, form a new digraph G_s by identifying $s = s_1$, $t_1 = s_2$ and $t = t_2$ *(series composition).*

TTSP digraphs are obviously acyclic (see figure 1).

[Figure 1]

By augmenting the set of operations as follows, we obtain the class of *looped SP digraphs* [1].

- Any TTSP digraph is a looped SP digraph.
- Given two looped SP digraphs G_1 and G_2 with terminals s_1, t_1, s_2, and t_2, form a new digraph G_r by identifying $s = s_1 = t_2$ and $t = t_1 = s_2$ *(parallel reverse composition).*

Figure 2 exhibits a looped SP digraph.

[Figure 2]

The other class of SP digraphs we consider in this paper consists of *Minimal Series Parallel Digraphs* [32] (in short *MSP* digraphs) which are inductively defined as follows.

- The graph with one vertex and no edges is MSP.
- If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are MSP, then so is $G_p = (V_1 \cup V_2, E_1 \cup E_2)$ *(parallel composition).*
- If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are MSP, then so is $G_s = (V_1 \cup V_2, E_1 \cup E_2 \cup (T_1 \times S_2))$, where T_1 is the set of sinks in G_1 and S_2 is the set of sources in G_2 *(series composition).*

General series parallel digraphs [32] (in short *GSP* digraphs) are the graphs whose transitive reduction [3] is MSP. Figure 3 shows a GSP digraph.

[Figure 3]

Any digraph considered above can be represented by means of a *decomposition tree* [32]. Each leaf of the tree corresponds either to a vertex (in case of MSP and GSP digraphs) or to an edge (in case of TTSP and looped SP digraph) in the original digraph. An internal node v is labeled S or P depending on the series or parallel composition of the digraphs represented by the subtrees rooted at the children of v. Trees representing looped SP digraphs have also nodes labeled R when a parallel reverse composition takes place. Nodes labeled S, P and R will be referred to respectively as S-nodes, P-nodes and R-nodes. Figure 4 shows an MSP digraph and a corresponding decomposition tree.

[Figure 4]

Notice that there can be several decomposition trees corresponding to the same MSP digraph, according to the fact that there can be different sequence of series and parallel compositions which give rise to the same MSP digraph.

In the remainder of the paper we first consider the problem of dynamically maintaining a collection of GSP digraphs under an arbitrary sequence of operations of the following kind.

- *create*(G, x) : return a GSP digraph G consisting of vertex x and no edges.
- *remove*(G) : delete from the collection the GSP digraph G. This operation assumes that G consists of one vertex and no edges.
- *series*(G_1, G_2, G) : given two GSP digraphs G_1 and G_2, return G as the series composition of G_1 and G_2. This operation destroys G_1 and G_2.
- *parallel*(G_1, G_2, G) : given two GSP digraphs G_1 and G_2, return G as the parallel composition of G_1 and G_2. This operation destroys G_1 and G_2.
- *insert*(x, y) : if there is a path from vertex x to vertex y, then insert the edge (x, y).
- *undo* : undo the last update operation (i.e., series, parallel or insert) not yet undone.
- *query*(x, y) : return *true* if there is a path from x to y; return *false* otherwise.
- *report*(x, y) : return an arbitrarily chosen path from x to y, if one exists.
- *reg-expr*(x, y) : return a regular expression which describes the set of all the paths between vertices x and y.

The update operations form a complete set, in the sense that any GSP digraph can be assembled or disassembled by a suitable sequence of such operations.

In the following section we present a data structure which supports each of the previous operations in the claimed time bounds. In particular, we will show how to support each create, remove and insert operation in $O(1)$ time and each series, parallel, insert, undo and query in $O(\log n)$ time. Each report operation takes $O(k + \log n)$ time, where k is the length of the achieved path. Similarly, reg-expr(x, y) can be supported in $O(h + \log n)$ time, with h being the total number of vertices in paths from x to y. We also show that all the above operations can be correctly performed by using an implicit representation based upon decomposition trees.

The same data structure can be used to achieve the same bounds for TTSP dags. For looped SP digraphs, we will need different techniques due to the presence of cycles. In this case, we allow also a *reverse* operation, which given two looped SP digraphs composes them according to parallel reverse composition.

3 Dynamic GSP and TTSP dags

In this section we present a data structure which supports the repertoire of operations for GSP dags in the claimed time bounds. The same data structure can be used with very few changes for the dynamic maintenance of TTSP dags. Therefore, we will restrict our attention to GSP dags.

The ideas underlying the data structure are the following. First of all, instead of representing a GSP dag, we store the MSP dag which is its transitive reduction. This is not a restriction

since for the problem of dynamically maintaining information about paths a GSP or its transitive reduction are completely equivalent. Therefore, without any loss of generality, we will restrict our attention to MSP dags. Second, we do not store the MSP dags explicitly, but rather represent them using decomposition trees. Because we have to perform dynamic operations, decomposition trees are implemented by means of the dynamic trees of Sleator and Tarjan [26]. We recall here that dynamic trees are able to support the following operations (among many others) on a forest of trees.

- *link*(v, w) : add the edge (v, w), where v is a root and w is a node in a different tree. This combines two trees containing v and w into a new one.
- *cut*(v) : if v is not a root, delete the edge from v to its parent. This divides the tree containing v into two trees.
- *lca*(v, w) : compute the least common ancestor of v and w.

Each operation is supported in $O(\log n)$ worst-case time [26], where n is the total number of nodes in the forest.

We maintain decomposition trees in a such a way that each S-node has exactly two children, while each P-node can have more than two children. We refer to such a representation of the decomposition trees as *compact decomposition trees* (in short *cd*-trees). Assuming that an MSP is given by means of a pointer to the root of the corresponding *cd*-tree, we perform our operations as follows.

In order to carry out a create(G, x), we initialize a new *cd*-tree to be a leaf x and associate it to the newly created MSP dag G. On the other hand, a remove(G) is done by deleting the corresponding *cd*-tree which consists of only one node. Both these operations can be accomplished in constant time.

When a series(G_1, G_2, G) operation is to be performed, we create a new node r corresponding to G and label it with S. Then, we make the roots r_1 and r_2 of the trees corresponding to G_1 and G_2 respectively the left and the right child of r. This will maintain the invariant that each S-node has exactly two children. Since a series operation causes at most two link operations to be performed on dynamic trees plus the creation of a new node, it can be accomplished in $O(\log n)$ time.

When a parallel(G_1, G_2, G) operation is to be performed, we do the following. If both the tree root r_1 corresponding to G_1 and the tree root r_2 corresponding to G_2 are not P-nodes, then we make r_1 and r_2 respectively the left and the right child of a new node r corresponding to G and labeled P. If at least one of the two roots is a P-node, say r_1, then we make the other root r_2 child of r_1. The last rule, referred to as *compacting rule*, causes P-nodes to have more than two children. Again, a parallel operation will cause at most two link operations to be performed on dynamic trees plus the creation of a new node. This can be accomplished in $O(\log n)$ time.

To perform an insert(x, y), nothing has to be accomplished. In fact, if there was a path from x to y, then an edge (x, y) will not add any useful information for our problem. Otherwise the insert(x, y) itself is not supposed to do anything. Clearly, this takes $O(1)$ time.

To handle undo operations, an *operation stack* is maintained, which contains pointer(s) to nodes in the forest of *cd*-trees. When either a series operation or a parallel operation which introduces two edges is performed, we push onto the stack two pointers to the roots made non-root. When a parallel operation introduce one edge because of the compacting rule, we push onto the stack a pointer to the only root made non-root. When an insert operation is performed, a dummy record without any pointer is pushed onto the stack. Therefore, an undo operation is performed by first

popping the top record of the stack. Then the node(s) (if any) pointed to by this record are accessed. If there is just one node x to be accessed, then it corresponds to a parallel composition with compacting rule and therefore the edge leaving x has to be deleted. If the nodes to be accessed are two, say x and y, then both the two edges leaving x and y as well as the node they are entering have to be deleted. As a consequence, each undo involves at most two cut operations on dynamic trees plus the deletion of a node and therefore can be implemented in $O(\log n)$ time. Since the above operations are the only ones which modify our data structure, we are now able to prove the following invariants.

Lemma 1 *There exists a path from vertex x to y ($x \neq y$) if and only if the following two conditions hold :*

(i) $lca(x, y)$ in the cd-tree is defined and labeled S;

(ii) x and y are respectively in the left and right subtree rooted at $lca(x, y)$.

Proof : By induction on the number of operations performed. □

Lemma 2 *In any cd-tree each S-node has exactly two children, while among the children of every P-node there is either a leaf or an S-node.*

Proof : By induction on the number of operations performed. □

Because of lemma 1, a query(x, y) is performed by testing whether lca(x, y) is defined and labeled S. Using dynamic trees, this can be implemented in $O(\log n)$ time. Paths between a given pair of vertices x and y are reported as the following pseudo-code shows.

```
procedure report(x, y, T);
begin
1. if query(x, y) then begin
2.     initialize both the path T and a queue Q to ∅;
3.     v := lca(x, y);
4.     ascend from x to v inserting into Q all the right children of S-nodes entered from the left;
5.     descend from v to y inserting into Q all the left children of S-nodes leaved from the right;
7.     while Q ≠ ∅ do begin
8.         remove a vertex u from Q; compute(u, T);
       end;
   end;
end;


procedure compute(u, T);
begin
1. case u of
       leaf :
2.         T := T ∪ {u};
       S-node :
3.         let i and j denote respectively the left and right children of u;
4.         compute(i, T); compute(j, T);
```

```
      P-node :
5.        pick i, a non P-node child of u;
6.        compute(i, T);
   endcase;
end;
```

The procedure *report* first checks whether there is a path from x to y in the GSP dag by means of a query(x, y) operation, i.e. by checking whether lca(x, y) in a *cd*-tree exists and is labeled S. If there is no such a path, then *report* stops. Otherwise, let us denote by $left(x, y)$ and by $right(x, y)$ respectively the paths in the *cd*-tree between x and lca(x, y) and between y and lca(x, y). If a path from x to y exists, then part of the subtree rooted at $lca(x, y)$ and in between left(x, y) and right(x, y) is recursively visited with the following two rules:

(i) when an S-node v is visited, then both the two children of v are examined;

(ii) when a P-node v is visited, then a non P-node child of v is examined.

Because of lemma 2, the above rules are well defined.

As far as a reg-expr(x, y) operation is concerned, we apply the same technique of a report(x, y) operation, with the only difference that now all the subtree rooted at $lca(x, y)$ and in between left(x, y) and right(x, y) is visited. Therefore rule (ii) above is substituted by the following rule:

(ii) when a P-node is visited, then all of its children are examined.

We will prove the correctness of this approach as well its timing analysis in the next section.

4 Correctness and Time Complexity

Before proving the correctness of the data structure and its time complexity, we need the following technical lemma.

Lemma 3 *Procedures report and reg-expr correctly accomplish their task in an MSP dag.*

Proof : By induction on the number of operations performed. □

Theorem 1 *The data structure correctly implements any sequence of create, remove, series, parallel, insert, undo, query, report and reg-expr operations.*

Proof : The correctness of the create, remove, series, and parallel operations derives from the definition of decomposition tree. Insert operations are correctly performed since in any case we do not have to add any new information to the reachability in the GSP dag. The correctness of each undo is a consequence of how the operation stack is handled. Finally, report and reg-expr are implemented correctly because of lemma 3. □

We now turn to the time analysis of our algorithm. We notice that in the implementation of procedure *report(x,y)* we have still to specify how to traverse the paths in the *cd*-tree between x and lca(x, y) and between lca(x, y) and y (lines 5. and 6. of procedure report). This is crucial, since there can be as many as $O(n)$ nodes in these paths. Therefore, a trivial implementation of this traversal would infringe the desired bound for report. Before describing the details on how

to speed up the search on these paths, we need a few more terminology. For any GSP dag G we define a *canonical source* and a *canonical sink* chosen among the set of sources and sinks of G. The canonical source and canonical sink of G are associated to the root of the *cd*-tree corresponding to G. Since each subtree of the *cd*-tree defines a GSP dag, we can associate inductively a canonical source and a canonical sink to each node z of a *cd*-tree, as follows.

- If z is a leaf, then the canonical source (sink) of z is z itself.
- If z is an S-node, then the canonical source (sink) of z is the same as the canonical source (sink) of the left (right) subtree rooted at z.
- If z is a P-node, then the canonical source (sink) of z is the same as the canonical source (sink) of an arbitrarily chosen child w of z which is not a P-node. In this case, w is said to be the *canonical child* of z.

The correctness of this definition hinges on lemma 2 which states that each S-node has exactly two children and that among all the children of a P-node, there is always a non P-node. Given any *cd*-tree T corresponding to a GSP dag G, we define the *left (right) ancestor* of any leaf ℓ of T as the highest node w in T such that ℓ is the canonical sink (source) of w. By using these definitions, we are able to speed up the search on the path from x to lca(x, y), as follows. Assume that we are considering a node z in such a path. If z is the left child of an S-node v ($v \neq lca(x, y)$), then we have to insert the right sibling of z into the queue Q defined in procedure *report*. Otherwise, we would like to skip to the least ancestor of z which can possibly enjoy the above property. The previous definitions give us a shortcut to this node, which indeed can be located as the left ancestor of the canonical source of the parent of z. As a result, line 4. of procedure *report(x,y)* can be implemented as follows.

```
z := x;
while z is not a child of lca(x, y) do begin
   while z is the left child of an S-node different from lca(x, y) do begin
      insert the right sibling of z into Q;
      z := parent(z);
   end;
   if z is not a child of lca(x, y) then
      z := LeftAncestor(CanonicalSource(parent(z)));
end;
```

The search in the path from lca(x, y) to y (line 5. of procedure *report*) can be sped up in a similar way and therefore the details are omitted.

To perform the search efficiently, to each node z of a *cd*-tree we need to associate a pointer to its parent, a pointer to its canonical source and a pointer to its canonical sink. Moreover, depending on z, we have the following additional pointers. If z is a leaf, there are two pointers to the left and right ancestors of z. If z is a P-node, there is a pointer to the canonical child and a pointer to the list of the remaining children of z. Finally, if z is an S-node, there are two pointers to the left and right children of z. To maintain all these pointers we need to perform some bookkeeping during the different operations. Indeed, if a series operation combines two GSP dags G_1 and G_2, then the canonical source (sink) of the newly created root r will be the canonical source of G_1 (canonical sink of G_2). Moreover, r will be the new left (right) ancestor of its canonical source (sink). If a parallel operation is performed, we can have the following two cases. If no new root is created, then

no pointer will be affected. Otherwise, denote by r the newly created root. Because of lemma 2, r has a non P-node which will be chosen as the canonical child of r. The canonical source (sink) of r will then be set to the canonical source (sink) of the canonical child of r. Moreover, r will be the new left ancestor (right ancestor) of its canonical source (sink). In case of an undo operation which will delete a root r, we need only to update the left (right) ancestor of the canonical sink (source) of r. By using these pointers, the speed up achieved in the procedure *report* allows us to prove the following theorem.

Theorem 2 *The data structure is able to support each create, remove and insert in $O(1)$ time, each series, parallel, undo and query in $O(\log n)$ time. Each report operation takes $O(k + \log n)$ in returning a path of length k, while reg-exp(x, y) can be supported in $O(h + \log n)$ time, with h being the total number of vertices in paths from x to y. The total space required is $O(n)$.*

Proof : Create and remove operations require the initialization or the deletion of a singleton node in the forest of *cd*-trees and therefore can be accomplished in constant time. Each insert does nothing and therefore can be supported in $O(1)$ time. Series, parallel, undo and query operations can be performed in $O(\log n)$ since they require that at most a constant number of *link*, *cut* and *lca* operations be performed on dynamic trees and that a constant number of pointers be updated.

To bound the total time required by a report operation, we notice the following two facts. First, the total time spent by *report(x,y)* in the paths from x to lca(x, y) and from lca(x, y) to y is proportional to the total number of nodes inserted into the queue Q during this step. The latter is clearly bounded by the total number of nodes visited in the *cd*-tree. Second, the total number of nodes visited in the *cd*-tree can be analyzed as follows. Because of rule (ii) (line 5. in procedure *compute*), the number of P-nodes examined is bounded above by the total number of S-nodes and leaves examined. As a consequence, the number of S-nodes examined is no more than the number of leaves examined. Since when a leaf is considered, it is inserted into the path to be reported (line 2. of procedure *compute*), the total number of nodes considered in the *cd*-tree while reporting a path of length k is $O(k)$. Therefore, the total time required by a report operation is $O(k)$ plus the time required by a query operation. This gives the claimed bound.

The analysis of procedure *reg-exp* can be carried out in a very similar vein and therefore it has been omitted. □

5 Dynamic looped SP digraphs

We now show how to maintain a collection of looped SP digraphs under an intermixed sequence of the following operations.

- *create*(G, s, t) : return a looped SP digraph G consisting of one edge (s, t).
- *remove*(G) : delete from the collection the looped SP digraph G. This operation assumes that G consists of one edge.
- *series*(G_1, G_2, G) : given two looped SP digraphs G_1 and G_2, return G as the series composition of G_1 and G_2. This operation destroys G_1 and G_2.
- *parallel*(G_1, G_2, G) : given two looped SP digraphs G_1 and G_2, return G as the parallel composition of G_1 and G_2. This operation destroys G_1 and G_2.
- *reverse*(G_1, G_2, G) : given two looped SP digraphs G_1 and G_2, return G as the reverse parallel composition of G_1 and G_2. This operation destroys G_1 and G_2.

- *undo* : undo the last update operation (i.e., series, parallel or reverse) not yet undone.
- *query*(x, y) : return *true* if there is a path from x to y; return *false* otherwise.
- *report*(x, y) : return an arbitrarily chosen path from x to y, if one exists.

The update operations form a complete set, in the sense that any looped SP digraph can be assembled or disassembled by a suitable sequence of such operations.

The undo operations can be performed with the help of an auxiliary stack, as shown for GSP dags. Therefore, in the following we will omit the details of such operation in our description.

Very recently, Afrati [1] proposed a fast parallel algorithm to check connectivity in looped SP digraphs. She uses two data structures, a decomposition tree of the shrunk digraph (i.e., the digraph obtained by shrinking each strongly connected component into a single node) and the forest of decomposition trees of the strongly connected components. Again, it is possible to check whether there is a path from a vertex x to a vertex y, by performing least common ancestor queries either in the shrunk digraph or in the forest of strongly connected components [1]. In fact, although the leaves of decomposition trees are now corresponding to edges (not to vertices) of the SP digraph, in order to decide whether a vertex x reaches a vertex y, it suffices to decide whether an edge entering x reaches an edge leaving y. By using dynamic trees, each query operation can therefore be accomplished in $O(\log n)$ time. It is not difficult to see that these two data structures can be dynamically maintained at the cost of $O(\log n)$ per update during series, parallel, reverse and undo operations. The details are substantially the same as those given in the previous sections and there is only some extra bookkeeping due to reverse operations.

This gives a total of $O(1)$ time for create and remove operations, and $O(\log n)$ time for each series, parallel, reverse and query operations. However, given these only two data structures, it does not seem possible to trace out paths in less than $O(n)$ time. To achieve the $O(k + \log n)$ bound, where k is the length of the returned path, we need a novel technique based upon ear decompositions [33] recently developed by Eppstein [11].

An *ear decomposition* is a partition of the edges of a graph. A *(directed) ear* is a (directed) path in the graph. An ear is *open* if its two endpoints are different. The first vertex of a directed ear E will by denoted by *start*(E), while the last vertex will be denoted by *end*(E). An ear decomposition consists of a partition of the edges of a graph into an order sequence of ears $E_1, E_2, \ldots, E_q$, such that the endpoints of an ear E_i, $i > 1$, appear in a previous ear E_j, $j < i$, but such that the internal vertices of each ear did not appear in any other previous ear. An open ear decomposition contains only open ears. Given a digraph and an open ear decomposition $D = \{E_1, E_2, \ldots, E_q\}$, E_i is said to be *nested* in E_j if $j < i$ and both the endpoints of E_i appears in E_j. We denote the path in E_j between the two endpoints of E_i as the *nesting range* of E_i in E_j. An ear decomposition D is said to be *nested* if both the following conditions hold:

(1) for each $i > 1$, there exists a $j < i$ such that E_i is nested in E_j;

(2) no two nesting ranges in any ear cross each other.

Eppstein showed that TTSP undirected graphs admit a nested open ear decomposition [11]. His proof can be easily extended to the class of looped SP digraphs, and by using nested open ear decompositions of looped SP digraphs, we are able to report efficiently paths between pairs of vertices. Before showing how the information about a nested ear decomposition can be maintained during a sequence of series, parallel, reverse and undo operations and how it can be used to perform report operations in the claimed bounds, we need some more terminology.

Given two ears E_i and E_k nested in the same ear E_j, we say that E_k *dominates* E_i (or equivalently that E_i *is dominated by* E_k) if and only if the nesting range of E_i is properly included in the nesting range of E_k. E_k *directly dominates* E_i (E_i *is directly dominated by* E_k) if there is no other ear E_h nested in E_j which is dominating E_i and dominated by E_k. E_k is a *dominating ear* if it is not dominated by any other ear. All these definitions can be extended to the nesting ranges n_i and n_k of E_i and E_k in E_j. Therefore we will indifferently say that the nesting range n_k *(directly) dominates* the nesting range n_i and we will talk about *dominating ranges.* Figure 5 shows a looped SP digraph and an associated ear decomposition. Ear $\{1,2,6,8,11,12\}$ is not dominated by any other ear, while ear $\{11,10,9,6\}$ is nested in $\{1,2,6,8,11,12\}$ and is dominated by $\{11,5,4,2\}$.

[Figure 5]

We maintain information about a nested open ear decomposition as follows. Given a looped SP digraph with terminals s and t, we define as ear E_1 a path from s to t (E_1 is the only ear which is not nested in any other ear). We do not maintain explicitly the relative ordering of the ears in the sequence, but rather we maintain information about the ear nesting. In fact, we associate to each ear E_j a rooted tree $T(E_j)$ of all the ears nested in E_j. The root r_j of $T(E_j)$ corresponds to E_j. The children of r_j correspond to the dominating ears nested in E_j (the dominating ranges in E_i) and are sorted in a left-to-right fashion according to the increasing distance of their endpoints from $start(E_j)$. In other words, a vertex corresponding to E_i precedes a vertex corresponding to E_k in the left-to-right order if and only if $start(E_i)$ precedes $start(E_k)$ in the directed path E_j. Furthermore, if different ears share the same range in E_j, then there will be just one vertex in $T(E_j)$ corresponding to all of them. The definition of the tree $T(E_j)$ is recursively completed as follows. The children of a non-root vertex v of $T(E_j)$ corresponding to an ear E_v, correspond to all and only the ears directly dominated by E_v, again sorted in a left-to-right fashion according to their distance from $start(E_v)$. In each tree $T(E_j)$, the edges are directed from a child to the parent.

This definition is consistent, because the open ear decomposition is nested. Therefore, to each ear of the decomposition $D = \{E_1, E_2, \ldots, E_q\}$ a tree of nesting ranges is associated. Each ear E_i, $i > 1$, of the decomposition D correspond to two vertices r_i and v_i in two different trees of nesting ranges, defined as follows. Vertex r_i is the root of $T(E_i)$, the tree of nesting ranges associated to E_i. Furthermore, since D is nested, there is an ear E_j such that E_i is nested in E_j. Therefore, vertex v_i is defined as the internal vertex in the tree $T(E_j)$ associated to the nesting range of E_i in E_j. As a consequence of the definition of trees of nesting ranges, we have that if $E_i \neq E_k$ then $r_i \neq r_k$. On the other hand, it can be that $v_i = v_k$ if and only if E_i and E_k are nested in the same ear and share the same nesting range. As for E_1, v_1 is undefined.

The q trees of nesting ranges can be combined together to form a unique rooted tree describing the nested ear decomposition, by joining the trees along the $q-1$ directed edges (r_i, v_i), $2 \leq i \leq q$. The root of the resulting tree will therefore be r_1, the root of the tree of nesting ranges associated to E_1. In order to distinguish the two different types of edges, we refer to the edges (r_i, v_i), $2 \leq i \leq q$, as *dashed* edges and to the edges in the trees of nesting ranges as *solid* edges. The global tree obtained is referred to as the *nested ear decomposition tree.* Using this global tree, a path between two vertices x and y can be traced according to the following steps.

1. Perform a query(x, y) operation. If there is no path from x to y, then stop.

2. Locate E_x and E_y, the ears in the decomposition containing respectively x and y. Let us denote by r_x and r_y the vertices in the nested ear decomposition tree corresponding respectively to E_x and E_y.

3. Starting from r_x and r_y and alternating between them one at the time, we climb towards the root of the nested ear decomposition tree. According to the traversal of the nested ear decomposition tree, we trace out paths in the looped SP digraphs. The path from x is traced forward and the path from y is traced backwards. At the beginning, we follow the path from x (backwards from y) to the endpoint of E_x (E_y). Each time we go from a vertex u to its parent v along a solid edge in a tree of nesting ranges $T(E_j)$, we continue to trace the directed path in E_j from the current endpoint of E_u to an endpoint of E_v, respectively the ears corresponding to the ranges u and v. If the edge (u, v) is dashed, then we continue to trace the directed path of E_u up to one of its endpoints. In this traversal, we mark each visited vertex in the nested ear decomposition tree.

4. The process terminates when we reach the same vertex of the nested ear decomposition tree for the second time. This vertex will correspond to a range in a certain ear E_i, which was reached by tracing paths forward from x and backward from y. Denote by x' the vertex in E_i reached while tracing forward, and by y' the vertex in E_i reached while tracing backward. Notice that as a consequence of the tracing procedure, x' and y' will be the two different endpoints of the ear E_i. If E_i is directed from x' to y', then we join the achieved path from x to x', the fragment of E_i from x' to y' and the achieved path from y' to y; the whole path can be now returned. Otherwise, we keep on tracing a (forward) path from x' and a (backward) path from y' in E_i and check whether there is a ear which spans the same range so far considered in E_i but in the opposite direction.

As a consequence of the open ear decomposition being nested, the above procedure terminates and correctly returns a path from vertex x to vertex y. It is possible to maintain on line the nested ear decomposition tree while performing series, parallel, reverse and undo operations without infringing the cost of $O(\log n)$ time per operation. In fact, after a series(G_1, G_2, G) operation, the decomposition of G is obtained by joining the two non-nested ears in the decompositions corresponding to G_1 and G_2. As a consequence, we have to merge the children of the roots of the two nested ear decomposition trees corresponding to G_1 and G_2. If all the children u_i of a node v in the nested ear decomposition tree such that the edges (u_i, v) are solid, are doubly linked with only the first and last item pointing to v, the merging of the children list can be accomplished in constant time. This will not affect the time complexity of the report operation. In fact, although the climbing phase will require more time, it can be shown that it is still bounded by the number of edges traced out in the actual path. A parallel(G_1, G_2, G) or a reverse(G_1, G_2, G) operation merges the two nested ear decomposition trees T_1 and T_2 respectively corresponding to G_1 and G_2 by introducing the dashed edge (r_2, r_1), where r_1 and r_2 are the roots of T_1 and T_2. Some bookkeeping information is required to distinguish between a parallel and reverse operation. This requires an extra constant time. As a result, a path of length k can be reported in $O(k + \log n)$ time without increasing the time required for each update in the looped SP digraph. The above argument leads to the following theorem.

Theorem 3 *It is possible to dynamically maintain looped SP digraphs with the following bounds. Each create and remove requires $O(1)$ time, while each series, parallel, reverse, undo and query can be performed in $O(\log n)$ time. Each report operation takes $O(k + \log n)$ in returning a path of length k. The total space required is $O(n)$.*

6 Concluding Remarks

In this paper we have shown how to maintain dynamically GSP dags, TTSP dags and looped SP digraphs. We have introduced data structures for updating (by both inserting and deleting either a group of edges or vertices) GSP dags, TTSP dags and looped SP digraphs of m edges and n vertices in $O(\log n)$ time. The time required to check on line whether there is a path between two given vertices is $O(\log n)$, while a path of length k can be returned in $O(k + \log n)$ time. In case of GSP and TTSP dags, a regular expression describing the set of all paths between two vertices x and y can be reported in $O(h + \log n)$, where $h \leq n$ is the number of vertices contained in any path from x to y. By carefully exploiting the properties of series parallel digraphs, the space complexity of the data structures presented is $O(n)$.

There are several related and perhaps intriguing open questions. First of all, it seems worth of further investigation to study whether there are other classes of digraphs for which the best bound known for maintaining the transitive closure of general dynamic digraphs can be improved. Furthermore, few results are known about the dynamic maintenance of shortest paths and related problems [6, 12, 25]. All the algorithms so far presented are either not fully dynamic or far from being optimal. Is it possible to improve these bounds for special classes of graphs, such as planar st-digraphs or series parallel digraphs?

Acknowledgements

We would like to thank David Eppstein for many useful suggestions and discussions. We are also indebted to Giorgio Ausiello for his continuous support and encouragement.

References

[1] F. Afrati, An efficient parallel algorithm for directed reachability in series parallel graphs, manuscript 1988.

[2] F. Afrati, D. Goldin, and P. Kanellakis, Efficient parallelism for structured data: directed reachability in S-P dags, Technical Report, Brown University, 1988.

[3] A. V. Aho, M. R. Garey, and J. D. Ullman, The transitive reduction of a directed graph, *SIAM J. Comput.* 1 (1972), 131-137.

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

[5] M. Ajtai and R. Fagin, Reachability is harder for directed than for undirected graphs, *Proc. 29th Annual Symp. on Foundations of Computer Science*, 1988, 358-367.

[6] G. Ausiello, G. F. Italiano, A. Marchetti Spaccamela, and U. Nanni, On-line computation of minimal and maximal length paths, in preparation.

[7] G. Ausiello, A. Marchetti Spaccamela, and U. Nanni, Dynamic maintenance of paths and path expressions in graphs, *Proc. 1st Internat. Joint Conf. ISSAC 88 (Int. Symp. on Symbolic and Algebraic Computation) and AAECC 6 (6th Int. Conf. on Applied Algebra, Algebraic Algorithms and Error Correcting Codes)*, *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1989.

[8] M. W. Bern, E. L. Lawler, and A. L. Wong, Why certain subgraph computations require only linear time, *Proc. 26th Annual Symp. on Foundations of Computer Science*, 1985, 117–125.

[9] M. Burke, and B. G. Ryder, Incremental iterative data flow analysis algorithms, Technical Report LCSR-TR-96, Department of Computer Science, Rutgers University, August 1987.

[10] R. J. Duffin, Topology of series parallel networks, *Journal of Mathematical Analysis and Applications* 10 (1965), 303–318.

[11] D. Eppstein, Parallel recognition of series-parallel graphs, manuscript, 1989.

[12] S. Even, and H. Gazit, Updating distances in dynamic graphs, *Methods of Operations Research* 49 (1985), 371–387.

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

[14] G. N. Frederickson, Data structures for on-line updating of minimum spanning trees, *SIAM J. Comput.* 14 (1985), 781–798.

[15] N. Horspool, Incremental generation of LR parsers, Technical Report, Department of Computer Science, University of Victoria, March 1988.

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

[17] G. F. Italiano, Amortized efficiency of a path retrieval data structure, *Theoret. Comput. Sci.* 48 (1986), 273–281.

[18] G. F. Italiano, Finding paths and deleting edges in directed acyclic graphs, *Inform. Process. Lett.* 28 (1988), 5–11.

[19] T. Kikuno, N. Yoshida, and Y. Kakuda, A linear time algorithm for the domination number of a series parallel graph, *Disc. Appl. Math.* 5 (1983), 299–311.

[20] J. A. La Poutré, and J. van Leeuwen, Maintenance of transitive closure and transitive reduction of graphs, *Proc. International Workshop on Graph-Theoretic Concepts in Computer Science*, *Lecture Notes in Computer Science*, vol. 314, Springer-Verlag, Berlin, 1988, 106–120.

[21] E. L. Lawler, Sequencing jobs to minimize total weight completion time subject to precedence constraints, *Annals of Discrete Math.* 2 (1978), 75–90.

[22] C. L. Monma, and J. B. Sidney, Sequencing with series-parallel precedence constraints, *Math of Operations Research* 4, 215–224.

[23] F. P. Preparata, and R. Tamassia, Fully dynamic techniques for point location and transitive closure in planar structures, *Proc. 29th Annual Symp. on Foundations of Computer Science*, 1988, 558–567.

[24] J. H. Reif, A topological approach to dynamic graph connectivity, *Inform. Process. Lett.* 25 (1987), 65–70.

[25] H. Rohnert, A dynamization of the all pairs least cost path problem, *Proc. 2nd Annual Symp. on Theoretical Aspects of Computer Science* (STACS 85), *Lecture Notes in Computer Science*, vol. 182, Springer-Verlag, Berlin, 1985, 279–286.

[26] D. D. Sleator, and R. E. Tarjan, A data structure for dynamic trees, *J. Comput. System Sci.* 24 (1983), 362–381.

[27] K. Takamizawa, T. Nishizeki, and N. Saito, Linear time computability of combinatorial problems on series parallel graphs, *J. Assoc. Comput. Mach.* 29 (1982), 623–641.

[28] R. Tamassia, and F. P. Preparata, Dynamic maintenance of planar digraphs, with applications, Manuscript, 1988.

[29] R. E. Tarjan, *Data structures and network algorithms*, CBMS-NSF Regional Conference Series in Applied Mathematics, vol. 44, SIAM, 1983.

[30] R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Disc. Meth.* 6 (1985), 306–318.

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

[32] J. Valdes, R. E. Tarjan, and E. L. Lawler, The recognition of series parallel digraphs, *SIAM J. Comput.* 11 (1982), 298–313.

[33] H. Whitney, Non-separable and planar graphs, *Trans. Amer. Math. Soc.* 34 (1932), 339–362.

[34] D. M. Yellin, A dynamic transitive closure algorithm, Research Report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, February 1988.

[35] D. M. Yellin, and R. Strom, INC: a language for incremental computations, *Proc. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988, 115–124.

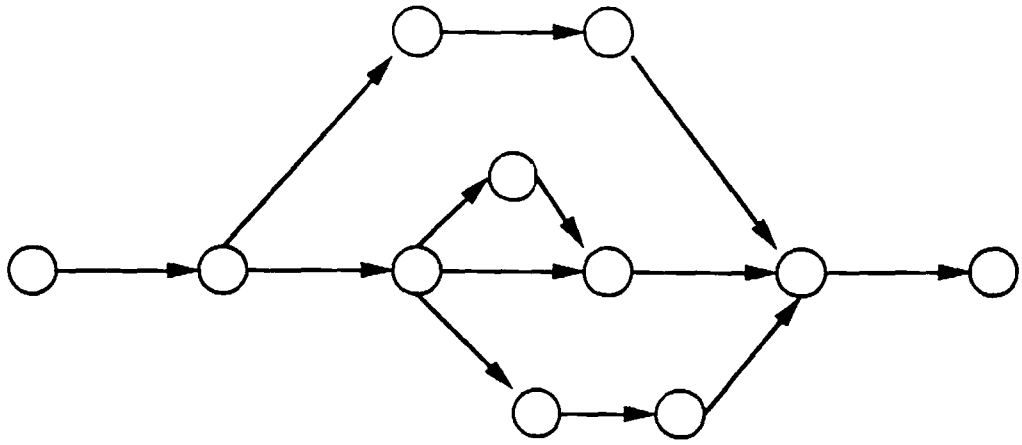

Figure 1: A TTSP dag.

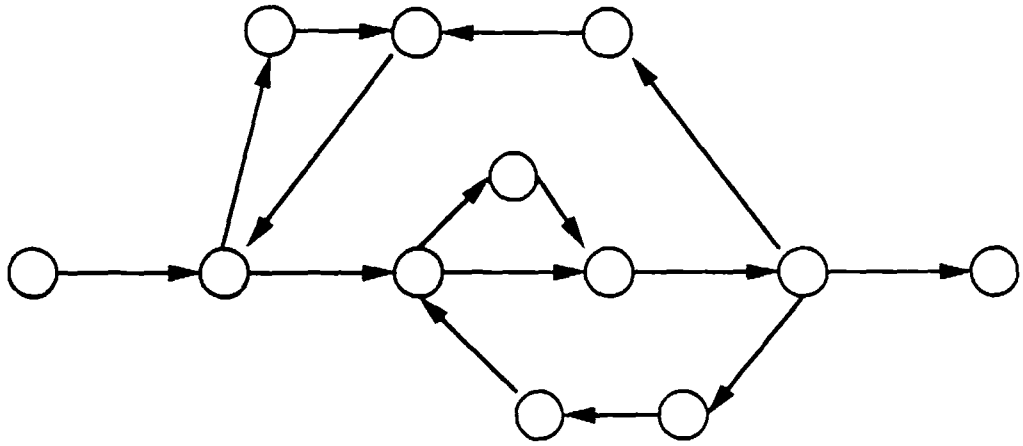

Figure 2: A looped SP digraph

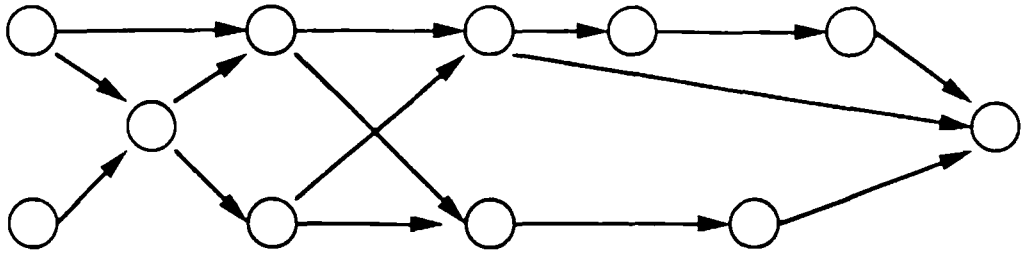

Figure 3: A GSP dag

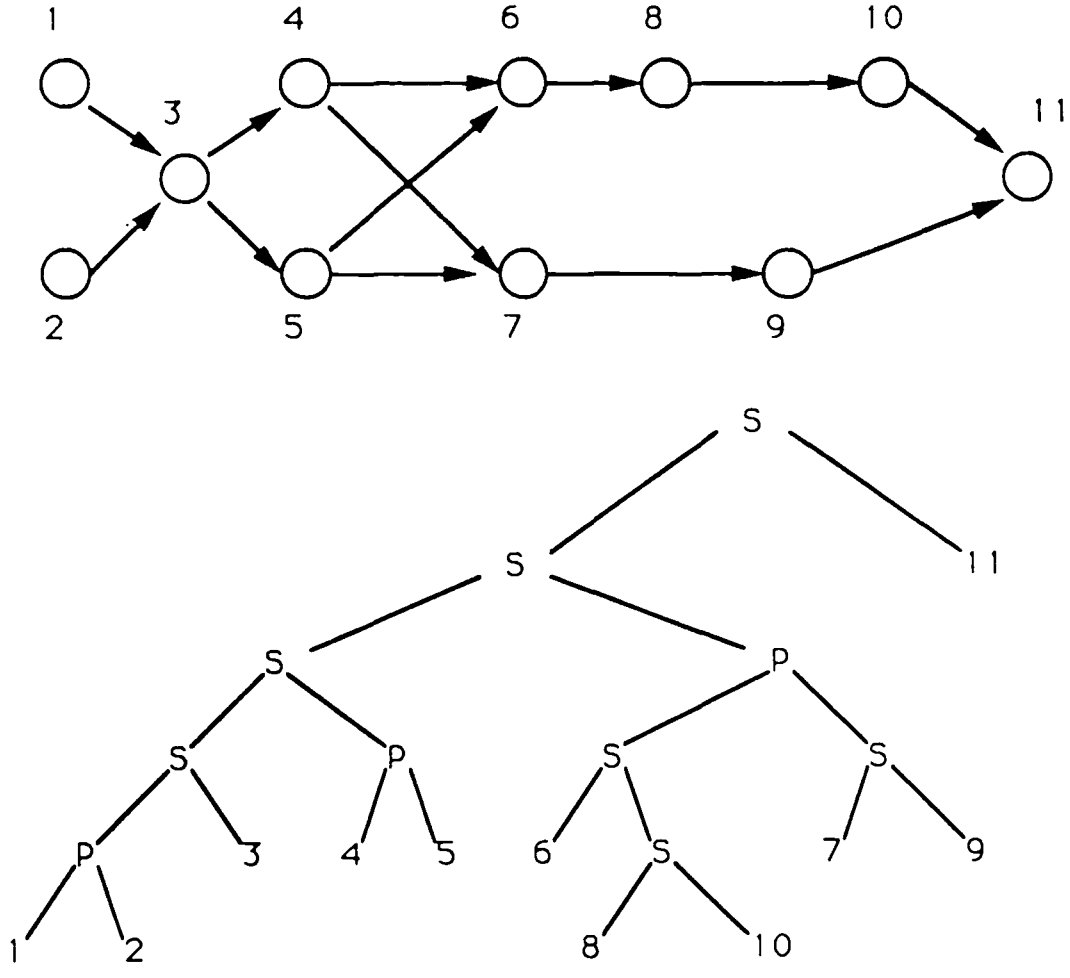

Figure 4 : An MSP dag and a decomposition tree

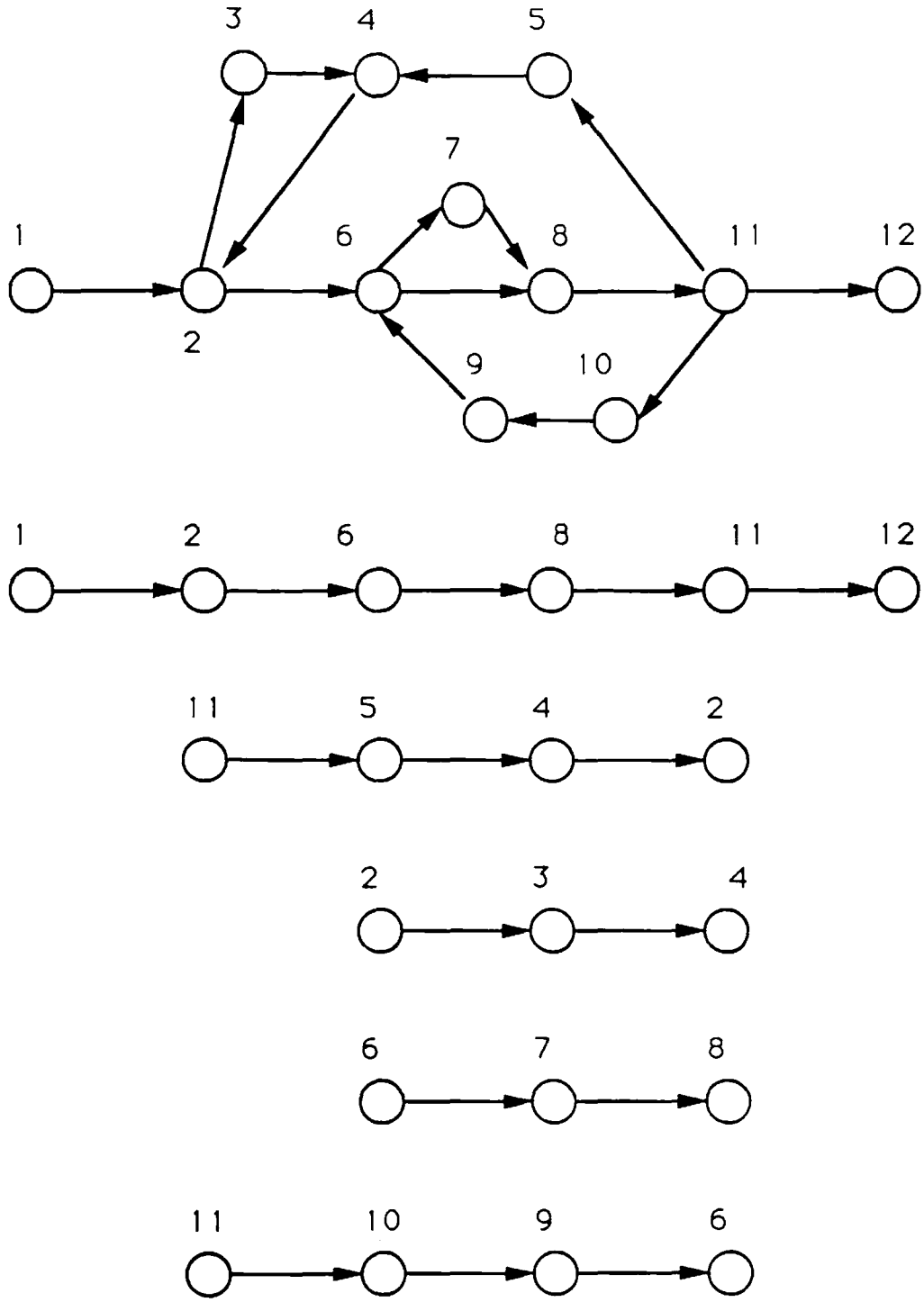

Figure 5 : A looped SP digraph and its nested ear decomposition