Theoretical Computer Science 72 (1990) 97-117
North-Holland

DYNAMIC MAINTENANCE OF DIRECTED HYPERGRAPHS[†]

Giorgio AUSIELLO* and Umberto NANNI**
Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy

Giuseppe F. ITALIANO***
Department of Computer Science, Columbia University, New York, NY 10027, USA

Abstract. In this paper we are concerned with the on-line maintenance of directed hypergraphs, a generalization of directed graphs previously introduced in the literature. In particular, we show how to maintain efficiently information about hyperpaths while new hyperarcs are inserted. We present a data structure which allows us to check whether there exists a hyperpath between an arbitrarily given pair of nodes in constant time and to return such a hyperpath in a time which is linear in its size. The total time required to maintain the data structure during the insertion of new hyperarcs is $O(mn)$, where m is the total size of the description of the hyperarcs and n is the number of nodes. This generalizes a previous result known for directed graphs and has applications in several areas of computer science, such as rewriting systems, database schemes, logic programming and problem solving. An extension of these results to hyperpaths between sets of nodes is also presented.

1. Introduction

Various kinds of hypergraphs have been extensively used in computer science as a suitable mathematical model to represent concepts and structures from different areas: rewriting systems; databases; logic programming; problem solving. In all cases, hypergraphs generalize the concept of graph in the sense that they consist of a set of nodes and a set of hyperedges (or hyperarcs) defined over the nodes.

The most widely used hypergraphs in computer science are *undirected hypergraphs*. In undirected hypergraphs (usually simply called hypergraphs [10]), hyperedges are arbitrary nonempty subsets of the set of nodes. An extensive use of hypergraphs has been made in database theory. In this application a hypergraph represents a database scheme and, in particular, the nodes correspond to "attributes" and the hyperedges correspond to "relation schemes". For example, several desirable

† Partially supported by the ESPRIT II Basic Research Action Program of the European Community under contract No. 3075 (Project ALCOM).
* Partially supported by the MPI Project on Theory and Design of Algorithms.
** Partially supported by SELENIA S.p.A.
*** Partially supported by NSF grants DCR-85-11713 and CCR-86-05353 and by an IBM Graduate Fellowship.

properties that a database must have (such as consistency of data or existence of efficient query answering strategies) have been expressed and studied in terms of hypergraph acyclicity [14].

A different model, which has been used in several applications, is *directed hypergraphs.* In a directed hypergraph a hyperarc is defined by a pair (X, i) where X is an arbitrary nonempty set of nodes and i is a node. Directed hypergraphs have a wide range of applications in computer science [11, 15, 22, 4, 7].

A first application arises in database theory. Given a set of attributes U, a set of *functional dependencies* is a relation over $P(U) \times U$, where $P(U)$ is the power set of U. A functional dependency (FD) from a set of attributes X to a single attribute i means that, given the values of all attributes in X, the value of attribute i is uniquely determined. Clearly a directed hypergraph may provide an immediate representation for such a relationship [4, 5]. In this context we might be interested to test if a given FD is derivable from a given set of FDs, or to derive from these all possible FDs computing their *closure.*

Another area in which hypergraphs provide a valuable representation is problem solving. Directed hypergraphs may be used in problem solving as an alternative to *and-or graphs*, for describing the relationship existing among a given problem P and the set of problems whose solution is required to solve P [19, 15].

Finally, another interesting application of directed hypergraphs arises in the representation and manipulation of *Horn formulae.* Among various classes of logical formulae, Horn formulae are particularly interesting in view of the fact that in Knowledge Based Systems [17] knowledge is often represented by means of *if*... *then*... clausal rules.

Also in the case of propositional Horn formulae, the use of directed hypergraphs is quite natural. In particular, each Horn clause corresponds to a hyperarc and testing the implication between propositional variables corresponds to checking the existence of a hyperpath between two nodes.

A particularly interesting case is when, in the process of building a Horn formula which represents our knowledge on a given domain by progressively adding new clauses, we want to check on-line the existence of a hyperpath from T to F (two special nodes corresponding to the truth values *true* and *false*), because such a hyperpath would imply the unsatisfiability of the whole formula [12, 7].

In this paper we provide data structures and algorithms for the problem of maintaining information about the connectivity of directed hypergraphs while new hyperarcs are inserted. In particular, we show how to check the presence of a hyperpath between a pair of nodes in constant time and to trace such hyperpaths with a cost which is linear in the length of the returned hyperpath.

The overall time required to maintain the data structure during the insertion of hyperarcs is $O(mn)$, where m is the total size of the description of the hyperarcs, and n is the number of nodes. This outperforms previously known algorithms for the same problem. In fact, the best known algorithms can require even $O(mn)$ to maintain the same information each time a new hyperarc is inserted. Moreover, these results are extended to the case in which we consider hyperpaths between sets of nodes.

The remainder of the paper consists of four sections. In Section 2 we give some preliminary definitions and basic results. Section 3 deals with the problem of maintaining on-line hyperpaths between any pair of nodes in the directed hypergraph. This result will be extended in Section 4 to the case of hyperpaths between two subsets of nodes. Section 5 contains some concluding remarks and open problems.

2. Basic definitions and representation of directed hypergraphs

A directed hypergraph (in the following referred to as dhg) is a generalization of the concept of directed graph. Given a set N of nodes, we will denote as $P(N)$ the power set of N.

Definition 2.1. A *directed hypergraph* $\mathscr{H}$ is a pair $\langle N, H\rangle$ where N is a set of *nodes* and H is a set of *hyperarcs*. Each hyperarc is an ordered pair (X, i) from an arbitrary nonempty set $X \in P(N)$ (*source set*) to a single node $i \in N$ (*target node*).

The basic parameters which will be taken into account in order to discuss the complexity of algorithms on directed hypergraphs are: the number of nodes (n), the number of hyperarcs (h), the number of source sets (n_c), the *source area*, that is the sum of cardinalities of all source sets (a), and the overall length of the description of the hypergraph ($m = |\mathscr{H}|$). If we assume to represent a directed hypergraph by means of adjacency lists we have that $m = a + h$. According to the same representation, the number of source sets is the same as the number of adjacency lists.

Definition 2.2 (Figs. 1, 2). Given a hypergraph $\mathscr{H} = \langle N, H\rangle$, a nonempty subset of nodes $X \subseteq N$ and a node $i \in N$, there is a (*directed*) *hyperpath* from X to i, if one of the following conditions holds:

(i) $i \in X$ (*extended reflexivity*), or

(ii) there is a hyperarc $(Y, i) \in H$ and for each node $j \in Y$ there exists a hyperpath from X to j (*extended transitivity*).

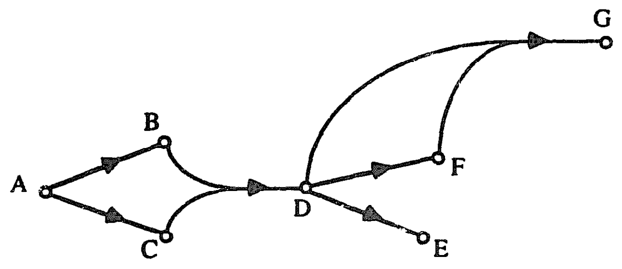

Fig. 1. An example of a hypergraph.

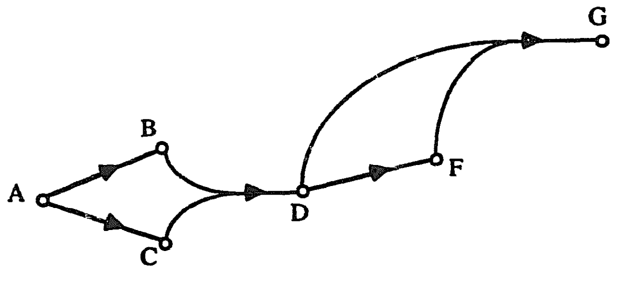

Fig. 2. A hyperpath from A to G in the hypergraph of Fig. 1.

Definition 2.3. Given a hypergraph $\mathcal{H}=\langle N, H\rangle$, the *closure* of $\mathcal{H}$, denoted as $\mathcal{H}^+$, is the hypergraph $\langle N, H^+\rangle$ such that (X, i) is in H^+ if and only if there exists in $\mathcal{H}$ a hyperpath from X to i. Given an arbitrary source set X of N, we call *closure of* X the set of nodes i in N such that (X, i) is in H^+.

On the basis of the preceding definitions we observe that the closure of a dhg may have size which is exponential in the number of nodes, simply because the source set X can be any element of $P(N)$.

In order to derive a structure which provides "almost" the same information as the closure of a dhg and in order to design efficient algorithms for the manipulation of directed hypergraphs, a data structure has been introduced for representing a dhg, which essentially is a labelled graph and which gives rise to a weaker notion of closure.

Definition 2.4. Given a hypergraph $\mathcal{H}=\langle N, H\rangle$, the *FD-graph* of $\mathcal{H}$ is the labelled graph $G(\mathcal{H})=\langle N_H, A_f, A_d\rangle$, where

$N_H = N \cup N_c$ is the set of nodes where N is called the set of *simple nodes* and $N_c=\{X\subseteq N \mid X$ is a source set in $\mathcal{H}\}$ is called the set of *compound nodes* (and each node in X will be called a *component* node of the compound node$X$);

$A_f \subseteq N_H\times N=\{(X, i)\mid$there exists a hyperarc (X, i) in $\mathcal{H}\}$ is the set of arcs referred as *full arcs*;

$A_d \subseteq N_c\times N=\{(X, j)\mid X\in N_c$ and $j\in X\}$ is the set of arcs referred as *dotted arcs*.

Figure 3 shows the FD-graph corresponding to the hypergraph in Fig. 1.

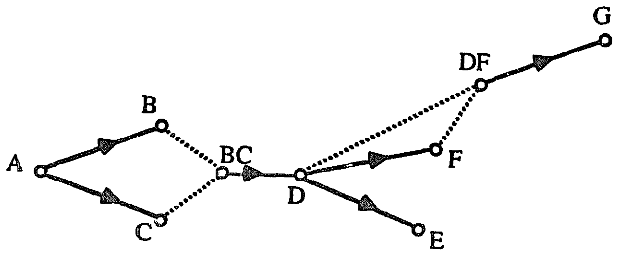

Fig. 3. The FD-graph corresponding to the hypergraph in Fig. 1.

Definition 2.5. Given an FD-graph $G(\mathscr{H}) = \langle N_H, A_f, A_d \rangle$, a nonempty subset of nodes $X \subseteq N_H$ and a node $i \in N_H$, an *FD-path* from X to i is a graph $G'(\mathscr{H}) = \langle N'_H, A'_f, A'_d \rangle$ which is a minimal subgraph of $G(\mathscr{H})$ and such that:

(i) $(X, i) \in A'_f \cup A'_d$, or

(ii) there is a simple node j, a full arc $(j, i) \in A'_f$ and an FD-path from X to j in $G'(\mathscr{H})$, or

(iii) there exists a compound node $Y \in H'_H$, a full arc $(Y, i) \in A'_f$ and, for each node $j \in Y$, we have $(Y, j) \in A'_d$ and there exists an FD-path from X to j in $G'(\mathscr{H})$.

Definition 2.6. Given an FD-graph $G(\mathscr{H}) = \langle N_H, A_f, A_d \rangle$ we define *FD-closure* of $G(\mathscr{H})$ the labelled directed graph $G^+(\mathscr{H}) = \langle N_H, A_f^+, A_d \rangle$, where A_f^+ contains:

(i) all the pairs (i, j) such that i is a simple node and there exists an FD-path in $G(\mathscr{H})$ from i to j, and

(ii) all the pairs (X, j) such that X is a compound node, $j \notin X$, and there exists an FD-path in $G(\mathscr{H})$ from X to j.

In [6] it is shown that, given a hypergraph $\mathscr{H}$ and the corresponding FD-graph $G(\mathscr{H})$, the FD-closure can be computed in $O(n_c \times |\mathscr{H}|)$, where $n_c < N_H$ denotes the number of source sets with cardinality greater than 1. Moreover, it is proved that a full arc (i, j) is in $G^+(\mathscr{H})$ if and only if the arc (i, j) is in $\mathscr{H}^+$.

Given a hypergraph $\mathscr{H}$, the FD-closure provides the closure of any set of nodes X such that either $|X| = 1$ (corresponding to simple nodes in the FD-graph) or X is the source set for some hyperarc in $\mathscr{H}$ (Fig. 4).

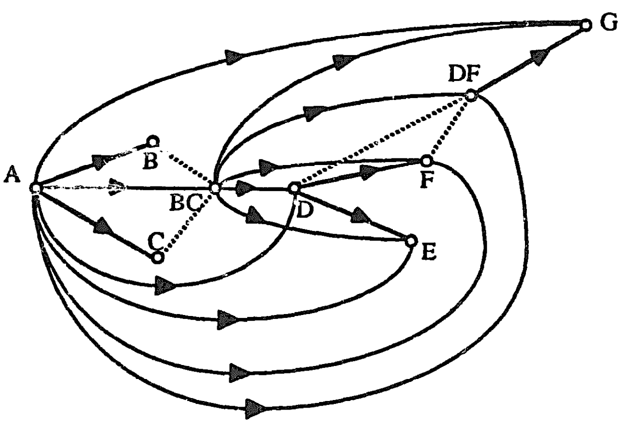

Fig. 4. The FD-closure of the FD-graph in Fig. 3.

Note that in order to manipulate dhgs efficiently in an on-line fashion, we need a different representation of FD-graphs. In fact, while in the case of the "static" problem we know all the simple and compound nodes of the hypergraph, when dynamic dhgs are considered, new compound nodes have also to be inserted in the data structure. If the number of new compound nodes is not known a priori and the order of insertions of hyperarcs in the structure is arbitrary, an efficient dynamic data structure for compound nodes has to be maintained. We will see how this is possible in the next section.

3. On-line traversal of dynamic directed hypergraphs

As we mentioned in the preceding paragraphs, in some applications the problem arises of maintaining the representation of a directed hypergraph while performing various operations *on-line*, such as inserting new hyperarcs and testing for the existence of a hyperpath between two nodes.

The study of on-line algorithms on dynamic graphs (that is graphs subject to both insertion and deletion of arcs) has been extensively developed in recent years [13, 20, 16, 18, 8], giving rise to algorithms and data structures with a good performance in terms of *amortized* costs [21]. In particular, great attention has been devoted to the semi-dynamic problem of maintaining information about the connectivity of the graph while inserting new arcs.

For example, in [16] a data structure is presented to maintain the transitive closure of a graph during the insertion of new arcs with an amortized time bound of $O(n)$ per insertion and total space $O(n^2)$, where n is the number of nodes in the graph. In [18] also, the transitive reduction [2] of a digraph is maintained within the same time and space bounds.

In [8] the problem of efficiently maintaining the minimal and maximal length paths between any pair of nodes on-line is tackled.

As we have already observed, the extension of these techniques to dynamic dhgs is not immediate because the number of compound nodes is not known: the FD-closure has been introduced to maintain the problem tractable.

The idea underlying the data structure is to maintain the FD-graph corresponding to the original directed hypergraph. In such a representation, a simple node can be accessed by means of full arcs only, while compound nodes are accessed by means of dotted arcs. In what follows, we shall deal only with FD-graphs and FD-paths, but every result may be directly formalizable in terms of directed hypergraphs and directed hyperpaths. When no danger of confusion arises, the two formalisms will be used interchangeably.

We will separately consider two situations corresponding to different requirements:

- first, we maintain the closure of any simple node;
- second, we maintain the FD-closure, i.e. the closure of any (simple or compound) node in the hypergraph.

In this section we introduce a data structure to deal with the first problem. In order to accomplish this task, we maintain an $n_s \times n_s$ array LAST, defined as follows.

LAST$[x, y]$ points to the last (simple or compound) node (except y) in an FD-path from the simple node x to the simple node y. If no such FD-path exists, then LAST$[x, y]$ has a special value *nil*.

With this additional information, the existence of an FD-path from a simple node i to a simple node j can be checked in constant time by examining the $[i, j]$th entry of the array LAST. If LAST$[i, j]$ is null, then there is no FD-path from node i to node j, otherwise an FD-path can be traced out by starting from j and proceeding in a depth first backward fashion as the following procedures show (a global variable hpath is used to collect the traversed full or dotted arcs in the required FD-path).

In the rest of this paper we will consider a dotted arc directed from the component node toward the compound node. This is done for the sake of uniformity of the pseudocode.

```
**procedure** hyperpath($i$, $j$: simple_node);
**var** hpath: set_of {full or dotted} arcs;
**begin**
  **if** LAST[$i$, $j$]⟨ ⟩**nil**
  **then begin**
    hpath := **null**;
    unmark all marked nodes (if any);
    mark $j$;
    FD-path($i$, $j$);
    **return** hpath
  **end**
**end**;
```

```
**procedure** FD-path($i$: simple_node, $j$: node); {$j$ is already marked}
**var** $w$: node;
**begin**
  **if** $i \neq j$
  **then** for each node $w$ in F-LAST($i$, $j$) **do**
  **begin**
    collect ⟨$w$, $j$⟩ in hpath;
    **if** $w$ is unmarked
    **then begin**
      mark $w$;
      FD-path($i$, $w$)
    **end**
  **end**
**end**;
```

The following function has been introduced to return a list containing the subsequent nodes to be scanned in the backward search:

```
**function** F-LAST($i$: simple_node, $j$: node): list of simple_node;
```

```
begin
  if $j$ is simple
  then return a list containing LAST[$i$, $j$]
  else return the list of components of $j$
end;
```

Notice that there is no need to store the node which has to be followed starting from a compound node during this backward search, since in this case in order to trace an FD-path, we are forced to go back to the simple nodes contained in the corresponding source set.

The invariants which we maintain for the array LAST will be such that a (full or dotted) arc can be examined at most once while tracing backward FD-paths. All the details of the method will be shown later on.

We now see how to maintain the information in the array LAST during the insertion of new hyperarcs in the original directed hypergraph. In order to accomplish this task, we associate an array $[1 \cdots n_s]$ referred to as REACH_y to each simple node y in N_s. In the course of the algorithm, the following invariant is maintained:

$$\text{REACH\_}y[x] = \begin{cases} 0, & \text{if there is an FD-path in the FD-graph from} \\ & \text{the simple node } x \text{ to the simple node } y, \\ 1, & \text{otherwise.} \end{cases}$$

The initialization of these arrays can be clearly accomplished in $O(n_s^2)$ time.

Note that the data structures for simple nodes are redundant, since REACH_$j[i] = 0$ if and only if LAST$[i, j] =$ nil. This is done for the sake of clarity and for a more uniform presentation of the algorithms. On the other side, this increases the space by no more than a constant factor and can be easily avoided in the implementation.

While dealing with dynamic hypergraphs we are allowed to introduce hyperarcs with arbitrary source sets. As a consequence we might have several hyperarcs with the same source set X. The compound nodes will be maintained in a balanced search tree referred to as T_c, while N_c denotes the set of compound nodes. We recall that balanced search trees are data structures which efficiently support search, insert and delete operations in logarithmic time [1, 9]. This will allow one to efficiently check whether the source set of the hyperarc to be introduced corresponds to a compound node already existent in the FD-graph. With this technique, only the necessary compound nodes will be introduced, thus making our representation nonredundant.

In the following we assume to deal with AVL trees and their basic operations, such as:

- AVL-search(item, tree), which returns either a pointer to the required item, or a special value *nil* if the item is not found;
- AVL-insert(item, tree) which inserts the item in the tree.

The array REACH of size n_s is also defined for compound nodes and the following invariant is maintained for each compound node x (with components $x_1, x_2, \ldots, x_q$)

and for each simple node i:

$$\text{REACH\_}x[i] = \sum_{(k=1,\ldots,q)} \{\text{REACH\_}x_k[i]\}.$$

That is, the entry REACH_$x[i]$ is equal to the number of simple nodes in X which are not reachable from the simple node i. Any array REACH_x is initialized when the compound node x and the array itself are created: this happens the first time we insert a hyperarc with source set X.

We implement FD-graphs by maintaining adjacency lists for each (simple or compound) node. In more detail, all the full [dotted] arcs leaving a node x are organized in the lists L_f(x) [L_d(x)]. Obviously L_d(x) will be empty for any compound node x.

We are now able to see how the closure of simple nodes can be maintained during the insertion of new hyperarcs from a given source set X to a simple node y.

We first introduce the function Compound which, given an arbitrary source set X, searches in the balanced tree T_c and returns the corresponding compound node x if it already exists, otherwise it is created and inserted in T_c, performing any necessary initialization.

```
function compound($X$: set of simple_node): compound_node;
var $x$: compound_node; $i, j$: simple_node;
begin
  $x$ := AVL_search($X$, $T_c$);
  if $x$ = null
  then begin
    create a new compound node $X$ pointed by $x$;
    AVL_insert($x$, $T_c$);
    for each {simple node} $i$ in $N_s$ do
      REACH_$x[i]$ := $\sum_{(j \in X)}$ REACH_$j[i]$;
    for each {simple_node} $i$ in $X$ do
      insert $x$ into L_d($i$)
  end;
  return $x$
end;
```

The procedure insert provides the required updates to the data structures while inserting a hyperarc from a source set X to a target node y:

```
procedure insert ($X$: set of simple_node, $y$: simple_node);
var $x$: node; $i$: simple_node;
begin
  if $|X| = 1$
  then $x$ := the element of $X$
  else $x$ := compound($X$);
  insert $y$ into L_f($x$);
  for each {simple_node} $i$ in $N_s$ do
    if REACH_$x$[$i$] = 0
    then closure ($i$, $x$, $y$)
end;
```

The aim of procedure closure is to update the arrays REACH and LAST after the insertion of the hyperarc (x, y). In particular, a closure(i, x, y) tries to find out the new connections that the inserted hyperarc has created for the simple node i by starting from node x. x is the node through which i reaches y and is useful in order to update the array LAST. The details are given in the following pseudocode.

```
procedure closure($i$: simple_node; $x$, $y$: nodes);
var $w$: node;
begin
  if REACH_$y$[$i$]⟨ ⟩0
  then begin
    REACH_$y$[$i$] := REACH_$y$[$i$] − 1;
    if REACH_$y$[$i$] = 0
    then begin
      if $y$ is a simple node
      then LAST[$i$, $y$] := $x$;
      for each $w$ in L_f($y$) ∪ L_d($y$) do
        closure($i$, $y$, $w$)
    end
  end
end;
```

Note that LAST$[i, y]$ is computed only when y is a simple node as should be done for the definition of LAST.

The correctness of this approach hinges on the following invariants.

Lemma 3.1. *After the insertion of any hyperarc, a node j is reachable from a simple node i if and only if* REACH_$j[i] = 0$.

Proof. *If-part*: We prove that at any time (except during the update) the following properties hold for the data structure:

(P1) If i and j are simple nodes and REACH_$j[i] = 0$, then there is an FD-path from i to j;

(P2) if x is a compound node with source set X, then the number of nodes in X not reachable from a simple node i does not exceed REACH_$x[i]$.

We proceed by induction on the number of (full or dotted) arcs examined during the execution of the procedure closure.

At the beginning, when the FD-graph is empty, properties (P1) and (P2) trivially hold since only REACH_$i[i]$ is initialized to 0 and no compound node exists. On the other hand, every compound node, when introduced, satisfies property (P2).

Assume now that properties (P1) and (P2) hold before examining an arc (x, y) during the execution of the procedure closure $(i, *, *)$ for any simple node i. Since (x, y) can be examined if and only if REACH_$x[i]$ has been set to 0, then one of the following conditions must be true:

(i) if (x, y) is a full arc, REACH_$y[i]$ will be set to 0 and there will be an FD-path from i to x due to the inductive hypothesis. This implies that there will also be an FD-path from i to y. Hence, properties (P1) and (P2) still hold.

(ii) if (x, y) is a dotted arc (i.e. x is a component node for y), then REACH_$y[i]$ is decremented by 1 but it is still greater than or equal to the number of simple nodes in y not reachable from i, since due to the inductive hypothesis x has become reachable from i. Thus properties (P1) and (P2) hold again.

Due to the arbitrary choice of the simple node i, this completes the if-part.

Only-if-part: We shall prove by induction on the number of inserted hyperarcs that:

(P3) If there is an FD-path from a simple node i to a simple node j, then REACH_$j[i]=0$.

Consider any simple node i. At the beginning (when the FD-graph is empty), no node is reachable from the node i except the node i itself. Since REACH_$i[i]$ is initialized to 0, the base of the induction holds.

Assume now that property (P3) holds before inserting a hyperarc $h=(x, y)$ from a source set $X=\{x_1, x_2, \ldots, x_q\}$ to y. We have the following two cases.

(i) If $\sum_{(k=1,\ldots,q)}\{$REACH_$x_k[i]\}$ is greater than 0 before the insertion of the hyperarc, then neither the nodes reachable from node i, nor the nodes j for which REACH_$j[i]$ is 0 can change after the insertion of the new hyperarc.

(ii) Consider now the case where $\sum_{(k=1,\ldots,q)}\{$REACH_$x_k[i]\}=0$ before the insertion of the hyperarc h and assume by contradiction that after inserting h there is a simple node w reachable from i but for which REACH_$w[i]$ is still greater than 0. Let us denote by H_w the hyperpath from i to w. Clearly H_w, although not unique, must include the newly introduced hyperarc h, otherwise the inductive hypothesis would be violated. Two cases may now arise.

(a) w coincides with y;

(b) w is different from y.

Case (a) is not possible, since REACH_$w[i]$ is set to 0 at the first call of the procedure closure$(i, *, *)$.

In case (b), let (V, w) be the last hyperarc of H_w from the source set $V=\{v_1, v_2, \ldots, v_p\}$ $(p \geq 1)$ to w and $H_1, H_2, \ldots, H_p$ be the p (proper) sub-hyperpaths of H_w from i to $v_1, v_2, \ldots, v_p$, respectively.

After the insertion of the hyperarc h, $\sum_{(k=1,\ldots,p)}$ REACH_$v_k[i]$ must be greater than 0, otherwise either $\sum_{(k=1,\ldots,p)}$ REACH_$v_k[i]$ was equal to 0 before the insertion of h (contradicting the inductive hypothesis) or the nodes v_k for which REACH_$v_k[i]$ was greater than 0 were forced to 0 after the insertion of the hyperarc (and therefore

REACH_$w[i]$ must also have been forced to 0). As a consequence, there must be (at least) a node v_k such that REACH_$v_k[i] > 0$ and it is reachable from i by means of the hyperpath H_k. Also in this case H_k must include the hyperarc h (otherwise the inductive hypothesis would be violated).

Since H_k is a proper sub-hyperpath of H_w, by repeating this reasoning with v_k in place of w, we will eventually reduce to the case (a) above in which $w = y$.

Due to the arbitrary choice of the node i, this completes the induction step and gives the thesis. □

Lemma 3.2. *After the insertion of any hyperarc, for any compound node w and any simple node i,* REACH_$w[i]$ *equals the number of simple nodes in the source set corresponding to w which are not reachable from i.*

Proof. The proof is similar to that of Lemma 3.1 and therefore has been omitted. □

Lemma 3.3. *After the insertion of any hyperarc, for any pair of simple nodes i and j such that there is an* FD-*path from i to j,* LAST$[i, j]$ *points to the node from which j has been reached for the first time from i during the execution of the procedure closure.*

Proof. Following the same proof given in Lemma 3.1, it can be proved that, after the insertion of any hyperarc, LAST$[i, j]$ is not null if and only if there is an FD-path from i to j. The first time a null LAST$[i, j]$ is given a non-null value by the procedure closure, we have that REACH_$j[i]$ has been decremented by 1 and set to 0. By Lemma 3.1, this implies that this is the first time that an FD-path from i to j has been discovered. □

Lemmas 3.1 and 3.2 assure that the closures of the simple nodes are correctly updated during the insertions of new hyperarcs, while Lemma 3.3 guarantees that the entries of the array LAST allow one to trace correctly an FD-path between any pair of simple nodes. In particular, the traced FD-path is the first FD-path which was established between that pair of nodes.

The following theorem summarizes the behaviour of the data structure under the insertion of new hyperarcs and provides an analysis of the costs involved.

Theorem 3.4. *Given an* FD-*graph H, with n* (*simple or compound*) *nodes and m* (*full or dotted*) *arcs in which hyperarcs are to be inserted in an on-line fashion, there exists a data structure which allows*:

(i) *to check whether there is an* FD-*path between any pair of simple nodes in constant time*;

(ii) *to return an* FD-*path* (*if one exists*) *between any pair of simple nodes in time linear in the length of the description of such an* FD-*path.*

The total time involved in maintaining the data structure while new hyperarcs are inserted is $O(mn)$. *The space required is* $O(m+n^2)$.

Proof. The time bounds on the FD-path queries are directly derived from the implementation of the data structure and from the fact that the procedure FD-path examines only (full or dotted) arcs which will be returned in the traced FD-path. The marks introduced guarantee that no such an arc can be examined more than once while constructing the FD-path.

As for the total time involved in maintaining the data structure, consider first the total time required to update the closure of a fixed node i (i.e. to update REACH_$j[i]$ for any j). In such a case, a (full or dotted) arc (x, y) may be scanned immediately after the insertion of new hyperarcs only if REACH_$x[i]$ was greater than 0. Once (x, y) has been scanned, REACH_$x[i]$ is permanently set to 0 and henceforth the arc (x, y) cannot be scanned again during subsequent calls of the procedure closure with last parameter i. As a consequence, maintaining the closure of each node requires $O(m)$ time and hence an $O(mn)$ overall time for all nodes derives.

We now need to consider the cost of maintaining the balanced tree which stores all the source sets introduced during the insertion of hyperarcs. Since at most n_c different source sets are introduced and for each introduced hyperarc we perform one search and at most one insert in this balanced tree, the total cost of maintaining this tree is $O(n_c \log n_c)$, which is dominated by $O(mn)$ since $n_c < m$ (each new hyperarc introduces one full arc), and $n_c < 2^{n_s}$.

The time required by procedure initialize is $O(n_s^2) = O(n^2)$. However, this preprocessing time can be reduced to $O(n)$ by initializing each entry of the arrays LAST and REACH the first time such an entry is accessed [3, p. 71].

The space complexity is a consequence of the storage utilization of the arrays REACH_j and LAST and of the FD-graph. □

4. Maintenance of FD-paths between sets of nodes

In Section 2 we have defined the closure of a generic set of simple nodes in a hypergraph. As we noted, the total number of different subsets is exponential in the number of simple nodes. In this section we restrict ourselves to the problem of maintaining the closure of compound nodes.

The overall time and space bounds are greater than in the case of maintaining the closure of simple nodes, due to the fact that now many operations on dynamic structures must be performed and connectivity information between compound nodes must be maintained. Nevertheless the cost of retrieving a hyperpath between an arbitrary pair of (simple or compound) nodes is again linear in the size of the exhibited hyperpath (except for an additive logarithmic cost depending on the total number of compound nodes).

The data structures for simple nodes remain nearly the same:

- an $n_s \times n_s$ array LAST, whose generic element LAST[i, j] contains a pointer to the last (simple or compound) node in a hyperpath from the simple node i to the simple node j;
- an $n_s \times n_s$ array REACH, whose generic element REACH[i, j] contains the value 0, if there is an FD-path from j to i, or 1 (otherwise).

For compound nodes we maintain the balanced search tree T_c using as a key the ordered set of components of each node, where the generic element (corresponding to the compound node w) contains the following additional information:

- the list of component nodes;
- the adjacency list L_f(w);
- the array LAST_TO: for any simple node i, LAST_TO[i] contains a pointer to the last (simple or compound) node in a hyperpath from w to i;
- the array REACH_TO: for any simple node i, REACH_TO[i] contains the value 0, if there is an FD-path from w to i, or 1 (otherwise);
- the array REACH_FROM: for any simple node i it contains a counter with the number of nodes in the set W not reachable from i;
- a balanced search tree C_REACH_FROM which for any compound node x contains the following information:
  VALUE: a counter with the number of nodes in the set W not reachable from x;
  NODE: a pointer to the compound node x inside the tree T_c.

The existence of an FD-path between two arbitrary nodes can be checked in O(log n_c) time if either of the nodes is compound, or in constant time if both the nodes are simple.

An FD-path can be retrieved in any case proceeding in a backward fashion starting from the second extreme (performing a search that requires O(log n_c) time if at least one of the nodes is not simple) and then with a time cost which is linear in the number of (full or dotted) arcs in the returned FD-path.

The following type and variable declaration for the main data structures will be useful to simplify the subsequent procedures.

```
type simple_node: integer;
     compound_node: record
        COMP: list of components;
        L_f: list of followers; {full arcs}
        LAST_TO: array [1 ··· $n_s$] of simple_node;
        REACH_TO: array [1 ··· $n_s$] of integer;
        REACH_FROM: array [1 ··· $n_s$] of integer;
        C_REACH_FROM: AVL_TREE of C_C_REACH
     end;
     C_C_REACH: record
        NODE: ^compound_node;
        VALUE: integer
     end;
```

```
var LAST: array [1 ⋯ $n_s$, 1 ⋯ $n_s$] of node;
    REACH: array [1 ⋯ $n_s$, 1 ⋯ $n_s$] of integer;
```

As usual, for any simple or compound node x, $L\_f(x)$ and $L\_d(x)$ denote the adjacency lists of full and dotted arcs, respectively. In order to simplify the pseudocode of the algorithm we assume to have two functions F-REACH and F-LAST, which are defined as follows. F-REACH(i, j), where i and j are (simple or compound nodes), returns an integer which is equal to 0 if and only if there is a hyperpath from i to j. Using the data structure defined above, this can be accomplished in O(log n) worst-case time. F-LAST(i, j), where i and j are (simple or compound nodes), returns a list of nodes which immediately precede j in a hyperpath from i to j. In more detail, this list of nodes is simply the source set of the last hyperarc in a hyperpath from i to j. The time required by the function F-LAST is linear in the length of the returned source set.

The procedure hyperpath takes two sets of simple nodes as arguments, returning a hyperpath between the corresponding simple or compound nodes.

```
procedure hyperpath($X$, $Y$: set of simple_node);
var $i$, $j$: node;
begin
  if $|X| = 1$ then $i := $ the element of $X$
  else $i :=$ compound($X$);
  if $|Y| = 1$ then $j :=$ the element of $Y$
  else $j :=$ compound($Y$);
  if F-REACH($i$, $j$) = 0
  then begin
    hpath := null;
    unmark all marked nodes (if any);
    mark $j$;
    FD-path($i$, $j$);
    return hpath
  end
end;
```

The procedure FD-path remains exactly the same as in the case of FD-paths between simple nodes (except for the arguments which can both be compound nodes).

We now describe how to maintain the data structure during the insertions of new hyperarcs. The function compound must initialize the possibly newly created node, computing the values of the various "REACH" entry, and propagating its closure toward its component nodes:

```
function compound($X$: set of simple_node): compound_node;
```

```
var x, w: compound_node;
   i, j: simple_node;
   z: C_C_REACH;
begin
   x := AVL_search(X, $T_c$);
   if x = null
   then begin
      create a new compound node X pointed by x;
      AVL_insert(x, $T_c$);
      for each {simple_node} i in X do
      begin
         x.REACH_FROM[i] := $\sum_{(j \in X)}$ REACH[i, j];
         insert x into L_d(i);
         x.REACH_TO[i] := 0;
         closure (x, x, i)
      end;
      for each {compound node} w in $N_c$ do
      begin
         create a new C_C_REACH record pointed by z;
            {it will be inserted into the tree x.REACH_FROM with key w}
         z.NODE := w;
         z.VALUE := $\sum_{(i \in X)}$ w.REACH_TO[i];
         AVL_insert (z, x.REACH_FROM);
      end
   end;
   return x
end;
```

The procedure insert should now be adapted as follows:

```
procedure insert(X: set of simple_node, y: simple_node);
var x: node; i: simple_node;
begin
   if |X| = 1
   then x := the element of X
   else x := compound(X);
   insert y into L_f(x);
   for each {simple_node} i in $N_s$ do
      if REACH_x[i] = 0
      then closure(i, x, y);
   case x of
      simple_node:
         for each w in $N_c$ do
```

```
          if w.REACH_TO[x] = 0
          then closure (w, x, y);
      compound_node:
        for each p in x.REACH_FROM do
          if p.VALUE = 0
          then closure (p.NODE, x, y)
end;
```

The procedure closure updates now the data in LAST and REACH during the insertion of the hyperarc (X, y). In particular, a closure(i, x, y) tries to find out the new connections that the inserted hyperarc has created for the (simple or compound) node i by starting from node x. x is the node through which i reaches y and is useful in order to update the array LAST. The details are given in the following pseudocode:

```
procedure closure(i, x, y: nodes);
var w: node; r: integer;
begin
  r := F-REACH(i, y);
  if r⟨⟩0
  then begin
    SET-REACH(i, y, r - 1);
    if r = 1 {that is: if F-REACH(i, y) = 0}
    then begin
      SET_LAST(i, x, y);
      for each w in L_f(y) union L_d(y)
        {L_d is empty for a compound node}
        do closure(i, y, w)
    end
  end
end;
```

The following procedures are introduced to simplify the update of REACH and LAST data:

```
procedure SET-REACH(i, j: node, value: integer);
var p: C_C_REACH;
begin
  if i is simple
  then if j is simple
    then REACH[i, j] := value
    else j.REACH_FROM[i] := value
  else if j is simple
```

```
        then i.REACH_TO_S[j] := value
        else begin
            p := AVL_search(i, j.REACH_FROM);
            p.VALUE := value
        end
end;

procedure SET-LAST(i, x, y:node);
    {x is the last node in a path from i to y}
    {if y is compound, no update is required}
begin
    if y is simple
    then if i is simple
        then LAST[i, y] := x
        else i.LAST_TO[y] := x
end;
```

Note that SET-LAST(i, x, y) makes the correct assignment only when y is a simple node as should be done for the definition of LAST.

It is now possible to prove theorems which are analogous to the results proved in the previous section. The only difference will be in the time bounds, due to the logarithmic costs introduced by balanced search trees. In particular, the correctness of the approach still hinges on the following invariants.

Lemma 4.1. *After the insertion of any hyperarc, a node j is reachable from a node i if and only if one of the following cases holds*:
- *i and j are both simple, and* REACH$(i, j) = 0$;
- *i is simple, j is compound, and* j.REACH_FROM$[i] = 0$;
- *i is compound, j is simple, and* i.REACH_TO$[j] = 0$;
- *i and j are both compound, and* j.REACH_FROM(i).VALUE $= 0$.

Lemma 4.2. *After the insertion of any hyperarc, for any compound node w and any node i,* REACH_$w(i)$ *equals the number of simple nodes in the source set corresponding to w which are not reachable from i.*

Lemma 4.3. *After the insertion of any hyperarc, for any node i and any simple node j such that there is an* FD*-path from i to j,* LAST_$j(i)$ *points to the node from which j has been reached for the first time from i during the execution of the procedure closure.*

As in the case of maintaining the closure of simple nodes, Lemmas 4.1 and 4.2 assure that the closures of any (simple or compound) node is correctly updated during the insertions of new hyperarcs, while Lemma 4.3 guarantees that the entries of the array LAST allow one to trace correctly an FD-path between any pair of nodes. In particular, the traced FD-path is the first FD-path which was established between that couple of nodes.

The overall time complexity of this algorithm can be characterized as follows.

Theorem 4.4. *Given an* FD-*graph* $G(\mathscr{H})$, *with* n *(simple or compound) nodes and* m *(full or dotted) arcs in which hyperarcs are to be inserted in an on-line fashion, there exists a data structure which allows:*

(i) *to check in constant time whether there is an* FD-*path between any couple of simple nodes.*

(ii) *to check in* $O(\log n)$ *time whether there is an* FD-*path between any couple of (simple or compound) nodes.*

(iii) *to return an* FD-*path (if one exists) between any pair of simple nodes in time* $O(k+\log n_c)$, *where* k *is the length of the description of such* FD-*path.*

The total time involved in maintaining the data structure while new hyperarcs are inserted is $O(mn \log n)$. *The space required is* $O(m+n^2)$.

Proof. The proof is a consequence of the preceding lemmas and follows the same line as the proof of Theorem 3.4, in which the logarithmic time to access the balanced search trees has been taken into account. □

The extra logarithmic times can be avoided by maintaining matrices instead of balanced search trees and by doubling their size when needed.

5. Conclusions

In this paper we have considered the problem of maintaining the transitive closure of directed hypergraphs in a dynamic environment. We have presented algorithms and data structures for the two different problems of maintaining either the closure of simple nodes or the closure of simple and compound nodes.

In the first case we showed how to perform the following operations:

- insertion of a hyperarc from a source set X to a simple target node i;
- checking the existence of a hyperpath between an arbitrary pair of simple nodes in constant time;
- returning a hyperpath between an arbitrary pair of simple nodes in a time which is linear in the size of the achieved hyperpath, that is in the size of the output.

The overall time required to update the data structure is $O(mn)$, where m is the size of the description of the (final) hypergraph, and n is the number of simple nodes (which is supposed to be given a priori). The space is $O(m+n^2)$. Note that if a simple directed graph is given as input, our algorithm has the same performance as the best known algorithms for this problem.

In the second case the following operations are allowed:

- insertion of a hyperarc from a source set X to a simple target node i;

- checking the existence of a hyperpath between an arbitrary pair of (simple or compound) nodes in $O(\log n_c)$ time;
- returning a hyperpath between an arbitrary pair of (simple or compound) nodes in $O(k+\log n_c)$ time, where k is the size of the description of the achieved hyperpath.

The overall time required to update the data structure is $O(mn \log n_c)$, where m is the size of the description of the (final) hypergraph, and n is the total number of simple and compound nodes.

There are several related and perhaps more intriguing problems. First, it seems worth investigating the case where deletions of hyperarcs are also allowed (provided the repeated insertion and deletion of the same hyperarc is not allowed). Second, our algorithms take $O(k)$ time to return a hyperpath of size k. Since k can be even exponential in the number of nodes, it seems important to select a hyperpath which enjoys certain minimality properties. Third, using tries instead of AVL-trees in Section 4 leads to a more space consuming data structure which allows one to trace out hyperpaths in optimal time. More precisely, the modified data structure requires $O(mn_s + n^2)$ space but allows one to return a hyperpath in time $O(|\text{input}|+|\text{output}|)$, thus getting rid of the extra logarithmic cost. The analysis of the trade-off between time and space complexity deserves further study.

References

[1] G.M. Adelson-Velskii and Y.M. Landis, An algorithm for the organization of information, *Soviet Math. Dokl.* **3** (1962) 1259-1262.

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

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

[4] G. Ausiello, A. D'Atri and D. Saccà, Graph algorithms for functional dependency manipulation, *J. ACM* **30** (1983) 752-766.

[5] G. Ausiello, A. D'Atri and D. Saccà, Strongly equivalent directed hypergraphs, in: *Analysis and Design of Algorithms for Combinatorial Problems*, Annals of Discrete Mathematics **25** (North-Holland, Amsterdam, 1985) 1-25.

[6] G. Ausiello, A. D'Atri and D. Saccà, Minimal representation of directed hypergraphs, *SIAM J. Comput.* **15** (1986) 418-431.

[7] G. Ausiello and G.F. Italiano, On-line algorithms for polynomially solvable satisfiability problems, *J. Logic Programming*, to appear.

[8] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela and U. Nanni, Incremental minimal length path, in: *Proc. 1st Ann. ACM-SIAM Symp. on Discrete Algorithms* (1990).

[9] R. Bayer and E. McCreight, Organization and maintenance of large ordered indices, *Acta Inform.* **1** (1972) 173-179.

[10] C. Berge, *Graphs and Hypergraphs* (North-Holland, Amsterdam, 1973).

[11] H. Boley, Directed recursive labelnode hypergraphs: A new representation language, *Artificial Intelligence* **9** (1977) 49-85.

[12] W.F. Dowling and J.H. Gallier, Linear-time algorithms for testing the satisfiability of propositional Horn formulae, *J. Logic Programming* **3** (1984) 267-284.

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

[14] R. Fagin, Acyclyc database schemes of various degrees: A painless introduction, in: *Proc. CAAP 83*, Lecture Notes in Computer Science **159** (Springer, Berlin, 1983) 65-89.

[15] S. Gnesi, U. Montanari and A. Martelli, Dynamic programming as graph searching: An algebraic approach, *J. ACM* **28** (1981) 737-751.

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

[17] R. Kowalski, *Logic for Problem Solving* (North-Holland, New York, 1979).

[18] J.A. La Poutré and J. Van Leeuwen, Maintenance of transitive closure and transitive reduction of graphs, Technical Report RUU-CS-87-25, Dept. of Computer Science, University of Utrecht, 1987.

[19] N.J. Nilsson, *Principles of Artificial Intelligence* (Springer, Berlin, 1982).

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

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

[22] M. Yannakakis, A theory of safe locking policies in database systems, *J. ACM* **29** (1982) 718-740.