01 Mar 1988

Deduction of a Functional Dependency from a Set of Functional Dependencies

James M. Richardson

Daniel C. St. Clair
Missouri University of Science and Technology

DEDUCTION OF A FUNCTIONAL DEPENDENCY FROM A SET OF FUNCTIONAL DEPENDENCIES

J. M. Richardson* and D. C. St. Clair

CSc-88-2

Department of Computer Science

University of Missouri-Rolla

Rolla, Missouri 65401 (314)341-4491

*This report is substantially the M.S. thesis of the first author, completed March, 1988.

ABSTRACT

This paper describes an algorithm called the Deduction Tracing Algorithm (DTA) which utilizes basic properties of functional dependencies from database systems and a modification of a tree search algorithm from artificial intelligence. The algorithm takes a set of functional dependencies, F, along with a specific functional dependency $L \rightarrow R$ as input and produces a list of functional dependencies from F that can be used to deduce $L \rightarrow R$. The resulting algorithm is easily automated to provide relational database users with a tool for organizing their queries.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

DEDUCTION OF A FUNCTIONAL DEPENDENCY FROM A SET OF FUNCTIONAL DEPENDENCIES

Jim Richardson
McDonnell Douglas
Information Systems Group
St. Louis, Missouri 63166

Daniel C. St. Clair
University of Missouri- Rolla
Graduate Engineering Center in St. Louis
St. Louis, Missouri 63385

Introduction

Relational database systems model functional dependencies between the attributes of various entities. While normalization is a common approach used in database design to reduce update and deletion anomalies,[1-3] it does not help the database user with one very fundamental problem. The problem consists of determining how to retrive information from the database when the functional dependency of interest is not contained in a single relation.

More specifically, suppose a relational database models a set of functional dependencies;

$$F = \{L_1 \rightarrow R_1, L_2 \rightarrow R_2, \ldots, L_k \rightarrow R_k\}$$

and that it is known that the functional dependency $L \rightarrow R$ is deducible from F. The problem is to determine how to use the elements of set F to deduce $L \rightarrow R$. The solution of this problem is a list of functional dependencies from F that can be used to deduce $L \rightarrow R$. For example, suppose $F = \{\mathbf{a} \rightarrow \mathbf{b}, \mathbf{b} \rightarrow \mathbf{c}\}$ and one desires to determine how $\mathbf{a} \rightarrow \mathbf{c}$ can be deduced from F. The deduction trace, the chain of functional dependencies which allow one to deduce **c** from **a**, is easy to see in this example. However, if F contains many complex functional dependencies with many different attributes, the trace will likely not be so obvious.

This paper describes an algorithm called the Deduction Tracing Algorithm (DTA) which utilizes the basic properties of functional dependencies from database systems and a modification of a tree search algorithm from artificial intelligence. The algorithm takes a set of functional dependencies, F, along with a specific functional dependency $L \rightarrow R$ as input and produces a list of functional dependencies from F that can be used to deduce $L \rightarrow R$. Before describing the DTA, it is important to establish several properties of functional dependencies and to describe the modified AND/OR search tree structure.

Properties of Functional Dependencies

The DTA is based on several fundamental properties of functional dependencies. For purposes of discussion, let U be a set of universal attributes and let F denote the set of functional dependencies;

$$F = \{L_1 \rightarrow R_1, L_2 \rightarrow R_2, ..., L_k \rightarrow R_k\}$$

where $L_i, R_i \subseteq U$ for i = 1,...k. Note that L_i, R_i contain elements of U which are connected by conjunctions. For example, if U = {**a**,**b**,**c**,**d**,**e**}, F might contain the functional dependency **ab** $\rightarrow$ **c**.

The process of deducing L $\rightarrow$ R from F is based on the set of inference rules known as Armstrong's axioms.[4] Assuming a set of attributes U and a set of functional dependencies F, Armstrong's Axioms describe how to move from one step of the deduction to the next.

Armstrong's Axioms:

Let X, Y, Z $\subseteq$ U,

1. (reflexivity). If Y $\subseteq$ X $\subseteq$ U then X $\rightarrow$ Y is logically implied by F.
2. (augmentation). If X $\rightarrow$ Y and Z $\subseteq$ U then XZ $\rightarrow$ YZ.
3. (transitivity). If X $\rightarrow$ Y and Y $\rightarrow$ Z then X $\rightarrow$ Z.

The technique for determining whether L $\rightarrow$ R can be deduced from F is based on the well-known concept of closure.[3,5]

Closure The closure of X with respect to F, denoted $C_F(X)$, is the set of attributes

$$A \subseteq U \ni X \rightarrow A \text{ can be deduced from F by Armstrong's axioms.}$$

If R $\subseteq C_F(L)$ then L $\rightarrow$ R can be deduced from F using Armstrong's Axioms. Hence, calculating $C_F(L)$ should be done first to determine whether or not L $\rightarrow$ R can be deduced from F. An algorithm for calculating $C_F(X)$ for set X has been described by Ullman.[5]

Closure Algorithm:

1. Let $X^{(0)} = X$
2. Calculate $X^{(i+1)} = X^{(i)} \cup \{A \mid \exists\ (Y \rightarrow Z) \in F,\ A \subseteq Z \text{ and } Y \subseteq X^{(i)}\}$
3. Repeat step 2 until $X^{(i)} = X^{(i+1)}$, at this point, $C_F(X) = X^{(i)}$.

Sets X and Y are said to be equivalent, $X \equiv Y$, iff $C_F(X) = C_F(Y)$.

It is now possible to formally define what it means to say a functional dependency can be deduced from a set of functional dependencies F.

Deducibility: A functional dependency $L \rightarrow R$ is deducible from (is logically implied by) a set of functional dependencies F iff

$$R \subseteq C_F(L).$$

One last property of functional dependencies must be established before the DTA can be described. The DTA requires that the set F be minimal. This is not a restriction of the DTA since Ullman proves that every set of functional dependencies F is equivalent to a set F' that is minimal.[5]

Minimal Set of Functional Dependencies: A set F of functional dependencies is minimal iff:

1. $\forall\ (L_i \rightarrow R_i) \in F$, R_i contains a single attribute, ie. $|R_i| = 1$.
2. $\neg\exists\ (X \rightarrow A) \in F\ \ni\ F - \{ X \rightarrow A \} \equiv F$. (Guarantees no redundant dependencies in F.)
3. $\neg\exists\ (X \rightarrow A) \in F$ and $Z \subseteq X\ \ni\ F - \{ X \rightarrow A \} \cup \{ Z \rightarrow A \} \equiv F$. (Guarantees no extraneous attributes.)

Reducing F to a minimal set F' requires use of the closure algorithm. Note that in item 1, there is no loss in having $|R| = 1$ as the definition of a functional dependence implies $\{\mathbf{ab \rightarrow cd}\} \equiv \{\mathbf{ab \rightarrow c, ab \rightarrow d}\}$.

Item 2 suggests how to remove redundant functional dependencies from F. For each L_i, $i = 1,...k$, find $C_D(L_i)$ where $D = F - \{L_i \rightarrow R_i\}$. If $R_i \subseteq C_D(L_i)$ the $L_i \rightarrow R_i$ is redundant and should be removed from F. For example, consider the set of functional dependencies:

$$F = \{\mathbf{a} \rightarrow \mathbf{b}, \mathbf{b} \rightarrow \mathbf{c}, \mathbf{ad} \rightarrow \mathbf{c}\}.$$

Remove $\mathbf{ad} \rightarrow \mathbf{c}$ from F and calculate $C_D(\{\mathbf{ad}\})$ where $D = F - \{\mathbf{ad} \rightarrow \mathbf{c}\}$. Using the closure algorithm,

$$X^{(0)} = \{\mathbf{a},\mathbf{d}\},$$

$$X^{(1)} = \{\mathbf{a},\mathbf{d},\mathbf{b}\}, \text{ and}$$

$$X^{(2)} = \{\mathbf{a},\mathbf{d},\mathbf{b},\mathbf{c}\} = C_D(\{\mathbf{ad}\}).$$

Since $\mathbf{c} \in C_D(\{\mathbf{ad}\})$, $\mathbf{ad} \rightarrow \mathbf{c}$ is redundant with respect to F and can be removed. In a normal case, all functional dependencies in F would be removed and checked in this same manner.

To insure that all left hand sides are minimal, proceed as suggested by item 3. For all combinations of attributes $A_j \subseteq L_i$ where $i = 1,...k$, $(L_i \rightarrow R_i) \in F$, calculate $C_F(A_j)$. If $R_i \subseteq C_F(A_j)$ for any A_j, then $L_i - A_j$ is extraneous and can be removed. Set F is modified so that $F = F - \{L_i \rightarrow R_i\} \cup \{A_j \rightarrow R_i\}$. This procedure is repeated for each functional dependency in F. For example if:

$$F = \{\mathbf{ab} \rightarrow \mathbf{c}, \mathbf{b} \rightarrow \mathbf{a}\},$$

start by taking $\mathbf{ab} \rightarrow \mathbf{c}$ and let $A_1 = \{\mathbf{b}\}$, i.e. remove **a** from the left hand side. Applying the closure algorithm gives $C_F(A_1) = \{\mathbf{b},\mathbf{a},\mathbf{c}\}$. Since $\{\mathbf{c}\} \subseteq C_F(A_1)$, **a** must be extraneous in $\mathbf{ab} \rightarrow \mathbf{c}$. Hence set F becomes

$$F - \{\mathbf{ab} \rightarrow \mathbf{c}\} \cup \{\mathbf{b} \rightarrow \mathbf{c}\} = \{\mathbf{b} \rightarrow \mathbf{c}, \mathbf{b} \rightarrow \mathbf{a}\}.$$

The following theorem provides a result used to reduce computation in the DTA.

Theorem: Given the functional dependency $L \rightarrow R$ where $R \not\subseteq L$ and the set of functional dependencies F, then $R \subseteq C_F(L)$ implies $\exists$ a unique integer $j \ni R \subseteq X^{(j)}$ but $R \not\subseteq X^{(j-1)}$ where $X^{(j)}$ is produced by the closure algorithm.

Proof: Let $R \subseteq C_F(L)$ and let $X^{(0)}, X^{(1)}, ..., X^{(s)} = C_F(X)$ denote the sets generated by the closure algorithm. Since in the closure algorithm,

$$X^{(i+1)} = X^{(i)} \cup \{A \mid \exists\ Y \rightarrow Z \in F,\ A \subseteq Z \text{ and } Y \subseteq X^{(i)}\}$$

for i = 0, ..., s-1, then $X^{(i)} \subseteq X^{(i+1)}$.

Since $R \not\subseteq L$, then $R \not\subseteq X^{(0)}$.

Further since $R \subseteq C_F(L) = X^{(s)}$, it must be the case that all elements in set R become elements of some set $X^{(j)}$. (While all elements of R may not have entered the sequence at the same step, the last group to enter the sequence did so at $X^{(j)}$.) Hence j is the value in [1..s] $\ni R \subseteq X^{(j)}$ but $R \not\subseteq X^{(j-1)}$

QED.

Combined AND/OR Trees

The search structure used in the Deduction Tracing Algorithm (DTA) is a combined AND/OR tree. This AND/OR tree is a variant of that described by Nilsson as a search structure used in artificial intelligence applications.[6]

Fig. 1 shows the basic structure of a combined AND/OR tree. The root node is denoted by *n*. Emanating from the root node are k_n descendent groups d_i. Each descendent group d_i contains one or more descendent nodes g_{ij} where j = 1,... $m_{d(i)}$. As the diagram shows, the individual descendent groups are related to each other disjunctively. The nodes within a single descendent group are related conjunctively.

The first group d_1, shown in **Fig. 1** contains only a single node g_{11}. Thus, if g_{11} is true then descendent group d_1 is true. In descendent group d_2, $m_{d(2)} = 2$ giving two nodes g_{21} and g_{22}. In this case both g_{21} and g_{22} must be true in order for descendent group d_2 to be true.

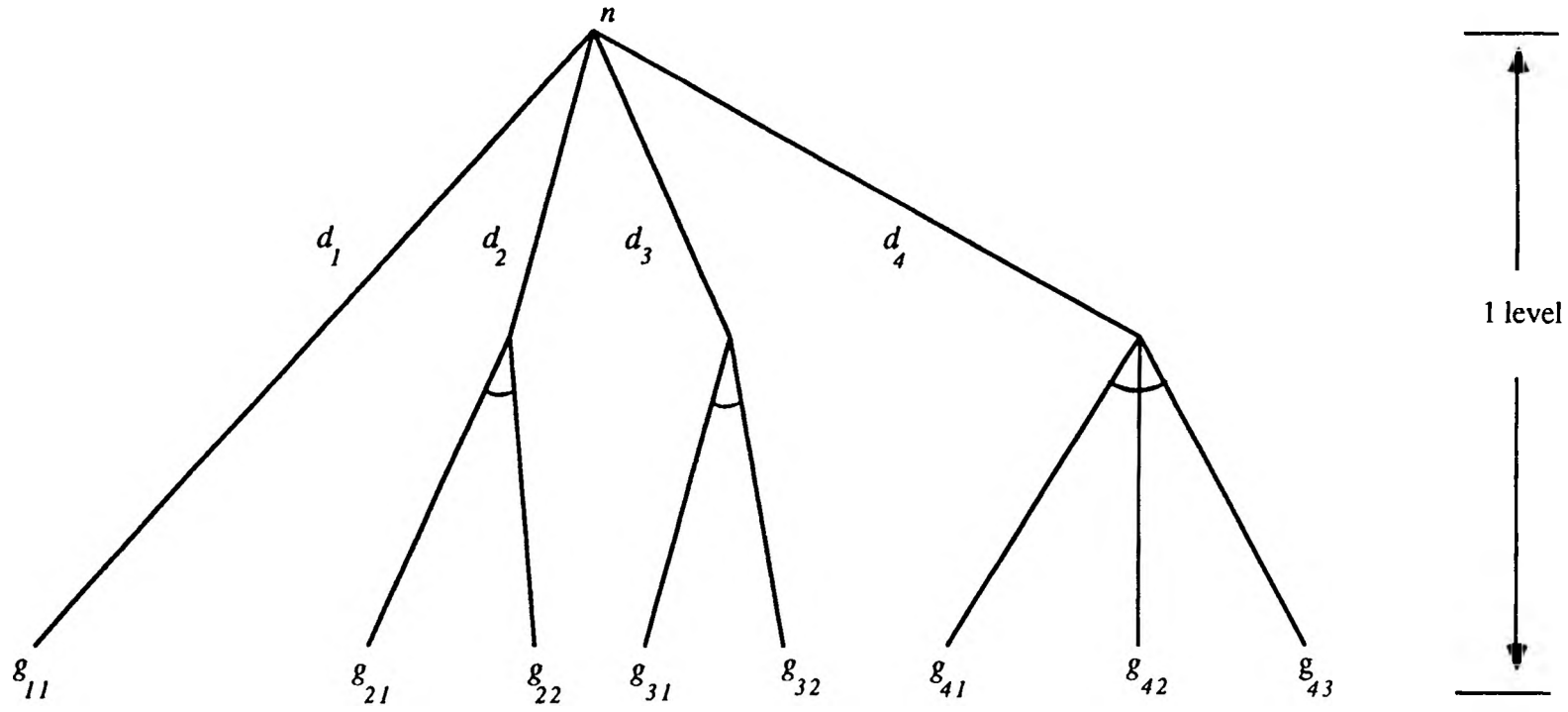

Fig. 1 Combined AND/OR trees

Fig. 2 illustrates how a combined AND/OR tree can be used to represent a set of functional dependencies, each having identical right hand sides. Consider the functional dependencies:

$$\mathbf{a} \rightarrow \mathbf{b}$$
$$\mathbf{cd} \rightarrow \mathbf{b}$$
$$\mathbf{ef} \rightarrow \mathbf{b}$$
$$\mathbf{ghk} \rightarrow \mathbf{b}$$

In this example, **b** is chosen as the root node. Note that **b** is the right hand side for all of the functional dependencies listed. For instance, as $\mathbf{a} \rightarrow \mathbf{b}$ contains only a single left hand side attribute, only **a** is needed to reach **b**. The descendent group d_1 represents this relationship. In the case of $\mathbf{cd} \rightarrow \mathbf{b}$ since attributes **c** and **d** are related conjunctively, both **c** and **d** are needed to reach **b**. This is shown in d_2.

If descendant groups d_1 - d_4 do not contribute to the the deduction trace, than node **b** is also not a part of the deduction trace. If all g_{ij} in at least one descendant group d_1 - d_4 contain goal attributes or are ancestors of descendant groups that contain goal attributes than **b** may be part of the deduction trace.

Deduction Tracing Algorithm

The Deduction Tracing Algorithm (DTA) seeks to determine how the functional dependency $L \rightarrow R$ can be deduced from a minimal set of functional dependencies F by searching a combined AND/OR tree. Without loss of generality, let R contain the single attribute $r \in U$. If R contained $k > 1$ attributes, the problem would be broken into k subproblems.[5]

The DTA requires two basic structures. The first is a representation for the combined AND/OR tree to be used to trace the dependencies. The second is a stack used to store each functional dependency as it is traced in the tree. When a path in the tree is developed that does not lead to part of the deduction, the functional dependencies from this path are popped from the stack. When the algorithm terminates, the complete trace is found, on the stack, in the order needed to deduce $L \rightarrow R$.

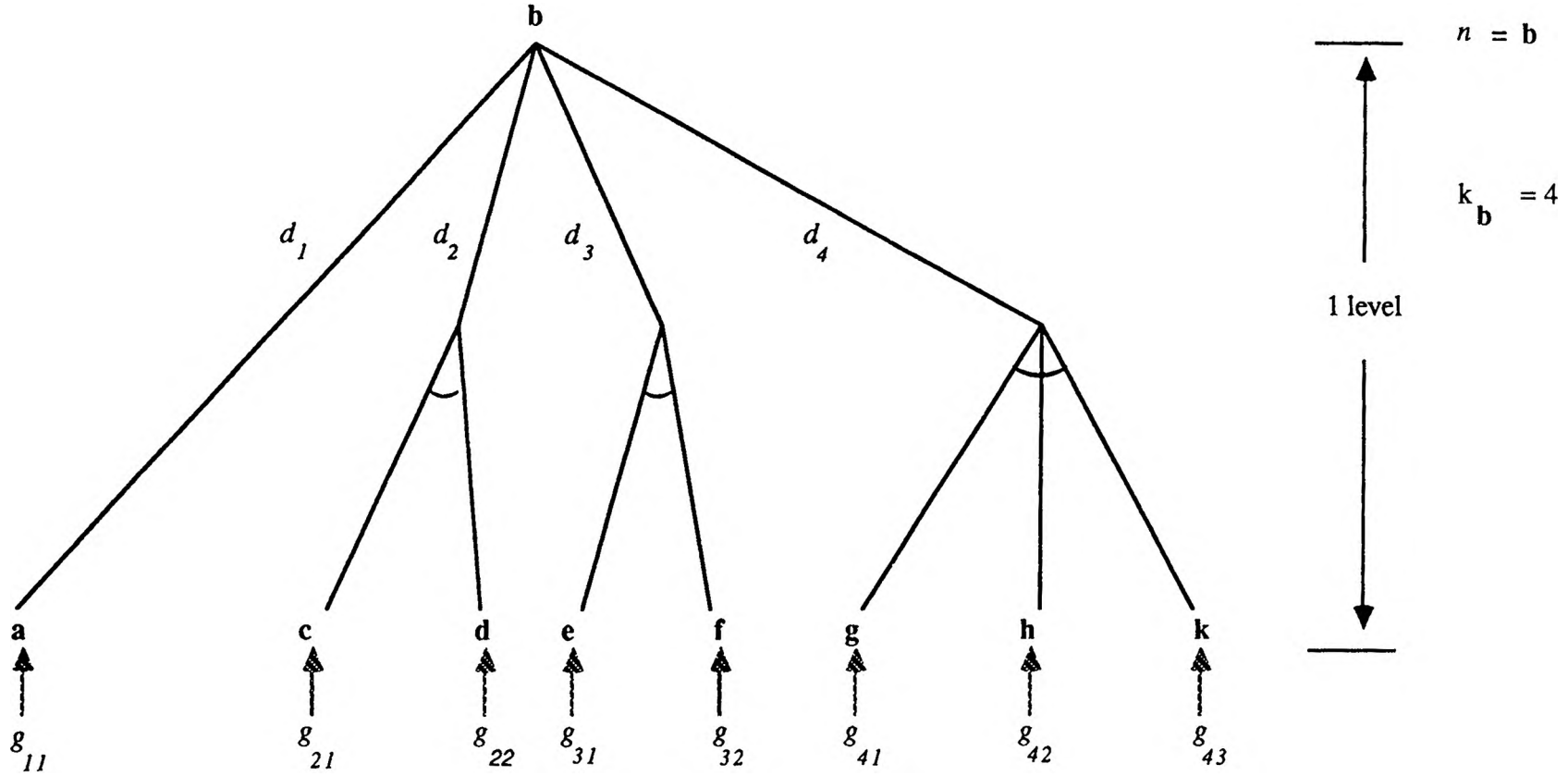

Fig. 2 Use of combined AND/OR tree to represent a set of functional dependencies.

The following description of the DTA uses n, d_i, and g_{ij} shown in **Fig. 1**. The functional dependency L → r is being deduced from the minimal set F.

1. If r ∈ C_F(L) then STOP, L → r can NOT be deduced from F.
   Else let s be the value of j from the Theorem where r ∈ $X^{(s)}$ and r ∉ $X^{(s-1)}$.
   The sets $X^{(s)}$ and $X^{(s-1)}$ are from the closure algorithm.

2. Remove any extraneous attributes from L using the technique previously described for producing minimal sets of functional dependencies.

3. Start by letting n = r be the root node of a modified AND/OR tree. Set c = 0 where c denotes the current level in the tree.

4. Let G = { $L_i \rightarrow n$ | $(L_i \rightarrow n) \in F$ }
   If G = { } then goto step 7.

5. Using the first $(L_i \rightarrow n) \in G$
   a) Create a branch d_J of the combined AND/OR tree. (This produces leaves g_{1j}, j = 1,...$m_{d(1)}$).
   b) Push $L_i \rightarrow n$ onto the stack.
   c) Set failed_path = false.
   d) G = G - {$L_i \rightarrow n$}
   e) Let $n = g_{11}$.
   f) c = c + 1.

6. While $n \notin$ L and c < s repeat steps 4 - 5.

7. If $n \notin$ L then failed_path = true.

8. a) If all attributes from L have been found and no partially resolved d_i remain, then STOP. The deduction trace is on the stack.
   Else
   Return to the ancestor node of n (i.e. n = ancestor node of current n)
   Let c = c - 1.

b) If failed_path then repeat

Pop the stack until the last functional dependency popped contains n on the right hand side.

9. If $g_{i,j+1}$ exists and not failed_path then

Let $n = g_{i,j+1}$

c = c + 1.

Else if G ≠ { } then select an $L_i \rightarrow n$ from G and create branch d_{i+1} from node n.

Push $L_i \rightarrow n$ onto stack

Set failed_path = false

Let $n = g_{i+1,1}$

c = c + 1.

Else

If n is a parent node of a goal node then set failed_path = false.

Repeat steps 8 - 9.

10. If c < s and $n \notin$ L then

repeat steps 4 - 7.

Else

repeat steps 7 - 9.

Example of the DTA

To illustrate the DTA, let

$$F = \{\mathbf{b \rightarrow a, dc \rightarrow b, f \rightarrow c, gf \rightarrow a, dr \rightarrow g, dz \rightarrow r, q \rightarrow z, z \rightarrow f}\}$$

be a given set of functional dependencies. It can be verified that F is minimal. Let

$$\mathbf{dq \rightarrow a}$$

be a functional dependency whose deduction trace is to be determined using the DTA.

In the first step, the calculation of $C_F(\{\mathbf{dq}\})$ gives the following results at each iteration of the closure routine:

$X^{(0)} = \{\mathbf{d},\mathbf{q}\}$

$X^{(1)} = \{\mathbf{d},\mathbf{q},\mathbf{z}\}$

$X^{(2)} = \{\mathbf{d},\mathbf{q},\mathbf{z},\mathbf{r},\mathbf{f}\}$

$X^{(3)} = \{\mathbf{d},\mathbf{q},\mathbf{z},\mathbf{r},\mathbf{f},\mathbf{c},\mathbf{g}\}$

$X^{(4)} = \{\mathbf{d},\mathbf{q},\mathbf{z},\mathbf{r},\mathbf{f},\mathbf{c},\mathbf{g},\mathbf{a}\}$

As can be seen, $\mathbf{a} \in X^{(4)}$. Thus, $\mathbf{a} \in C_F(\{\mathbf{dq}\})$ which implies that $\mathbf{dq} \rightarrow \mathbf{a}$ is deducible from F by Armstrong's Axioms. Since $\mathbf{a} \in X^{(4)}$ but $\mathbf{a} \notin X^{(3)}$, s = 4.

The next step is to remove any existing extraneous attributes from the left hand side of $\mathbf{dq} \rightarrow \mathbf{a}$. Since $\mathbf{a} \notin C_F(\{\mathbf{d}\}) = \{\mathbf{d}\}$ and $\mathbf{a} \notin C_F(\{\mathbf{q}\}) = \{\mathbf{q},\mathbf{z},\mathbf{f},\mathbf{c}\}$, there are no extraneous attributes in the left hand side of $\mathbf{dq} \rightarrow \mathbf{a}$.

Having determined that $\mathbf{dq} \rightarrow \mathbf{a}$ is deducible from F, step 3 of the algorithm can be performed by letting $n = \mathbf{a}$ and c = 0. This establishes **a** as the root node.

Step 4 is used to find all functional dependencies such that $L_i \rightarrow \mathbf{a}$. These functional dependencies are grouped into set $G = \{\mathbf{b} \rightarrow \mathbf{a}, \mathbf{gf} \rightarrow \mathbf{a}\}$. In step 5 branch d_1 is built from $\mathbf{b} \rightarrow \mathbf{a}$,

$\mathbf{b} \rightarrow \mathbf{a}$ is pushed on the stack,
failed_path = false,
$G = G - \{\mathbf{b} \rightarrow \mathbf{a}\} = \{\mathbf{gf} \rightarrow \mathbf{a}\}$,
$n = g_{11} = \mathbf{b}$, and
c = 1.

The tree and stack are shown in **FIG. 3.**

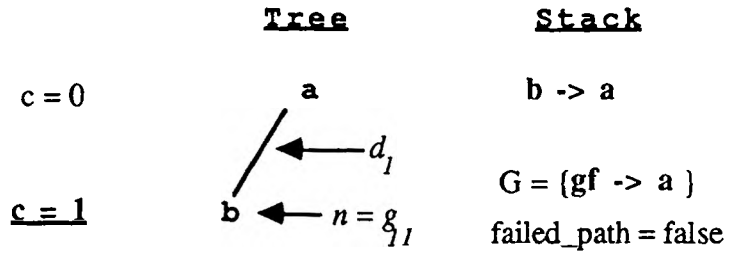

Fig. 3 DTA tree and stack prior to expansion at node b.

Since **b** $\notin$ {**dq**} and c = 1 < s = 4 as required in step 6, the process returns to step 4 to continue building the depth of the tree. With n = **b** and G = {**dc** $\rightarrow$ **b**}, branch d_1 is built from **dc** $\rightarrow$ **b** in step 5,

dc $\rightarrow$ **b** is pushed on the stack,
failed_path = false,
G = G - {**dc** $\rightarrow$ **b**} = { },
$n = g_{11}$ = **d**, and
c = 2.

The tree and stack are shown in **Fig. 4**.

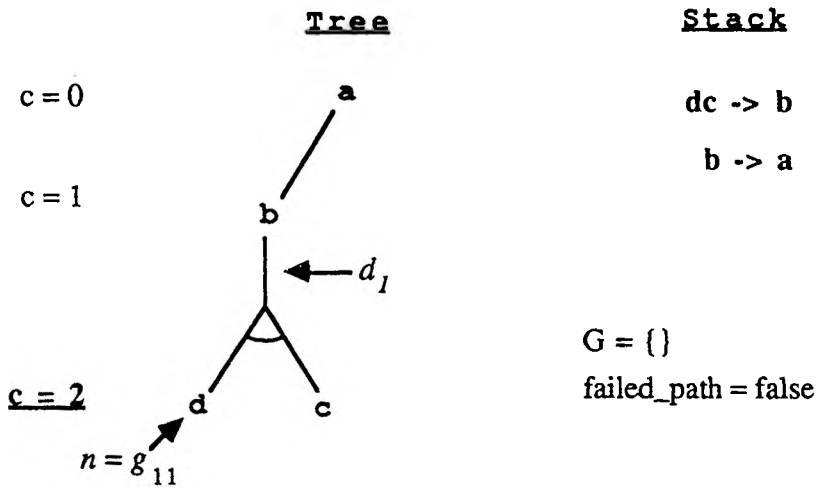

Fig. 4 Tree expansion stops at node d since d $\in$ {dq}.

Since n = **d** and **d** $\in$ {**dq**} in steps 6-7, the process moves to step 8 and starts developing the width of the tree at the current level 2. Not only does **q** need to be found, d_1 is still only partially resolved. Accordingly, n is moved back to the ancestor node of **d** and c is decremented to 1. The tree and stack are shown in **Fig. 5**.

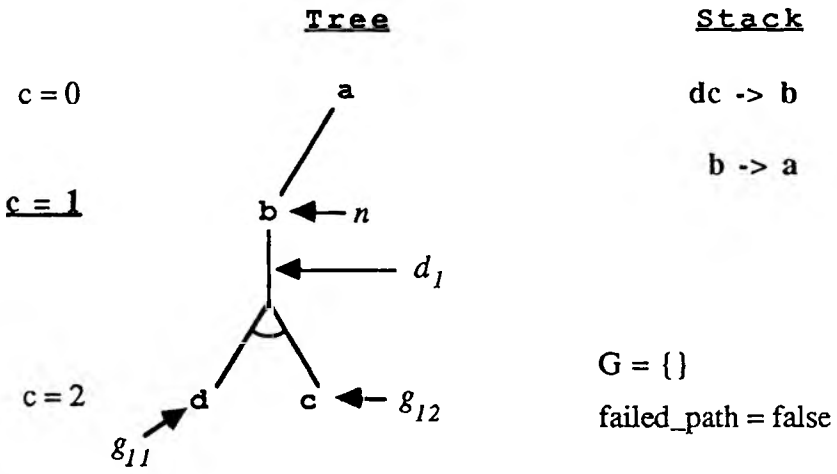

Fig. 5 To continue expanding the tree for dc → b, the DTA returns to node b.

In step 9, since g_{12} exists and not failed_path is true, $n = g_{12} = \mathbf{c}$. The counter c is incremented to 2 giving the tree shown in **Fig. 6**.

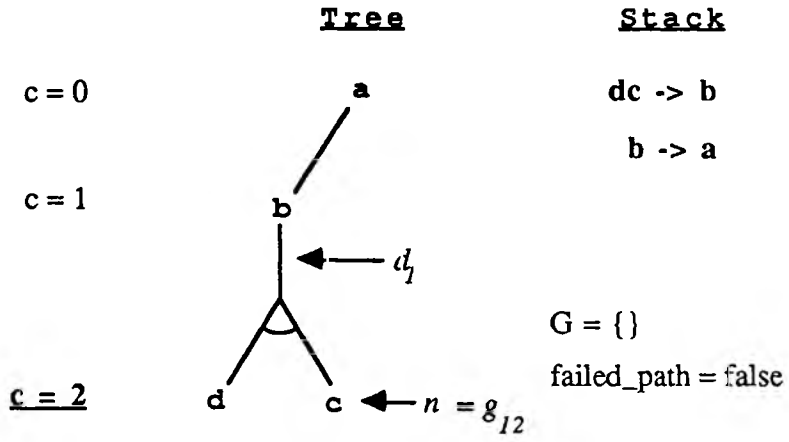

Fig. 6 DTA tree and stack prior to expansion at node c.

Since c = 2 < s = 4 and $\mathbf{c} \notin \{\mathbf{dq}\}$, step 10 instructs the process to return to step 4. The tree will now be expanded from the node $n = \mathbf{c}$. With $n = \mathbf{c}$ and $G = \{\mathbf{f} \rightarrow \mathbf{c}\}$, branch d_1 is built from $\mathbf{f} \rightarrow \mathbf{c}$ in step 5,

$\mathbf{f} \rightarrow \mathbf{c}$ is pushed on the stack,

G = G - $\{\mathbf{f} \rightarrow \mathbf{c}\}$ = { },

$n = g_{11}$ = f, and

c = 3.

The result is shown in **Fig. 7**.

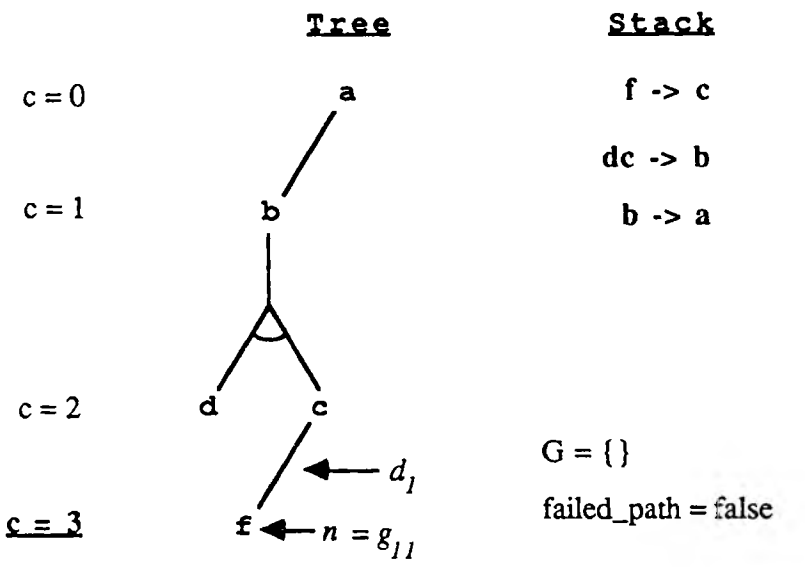

Fig. 7 Tree expanded from node c to node f using $\mathbf{f} \rightarrow \mathbf{c}$.

Since c = 3 < s = 4 and $\mathbf{f} \notin \{\mathbf{dq}\}$ as required in step 6, the process returns to step 4 and continues to search down the tree from n = f. With n = f and G = $\{\mathbf{z} \rightarrow \mathbf{f}\}$, branch d_1 is built from $\mathbf{z} \rightarrow \mathbf{f}$ in step 5,

$\mathbf{z} \rightarrow \mathbf{f}$ is pushed on the stack,

failed_path = false,

G = G - $\{\mathbf{z} \rightarrow \mathbf{f}\}$ = { },

$n = g_{11}$ = z, and

c = 4.

The tree and stack are shown in **Fig. 8**.

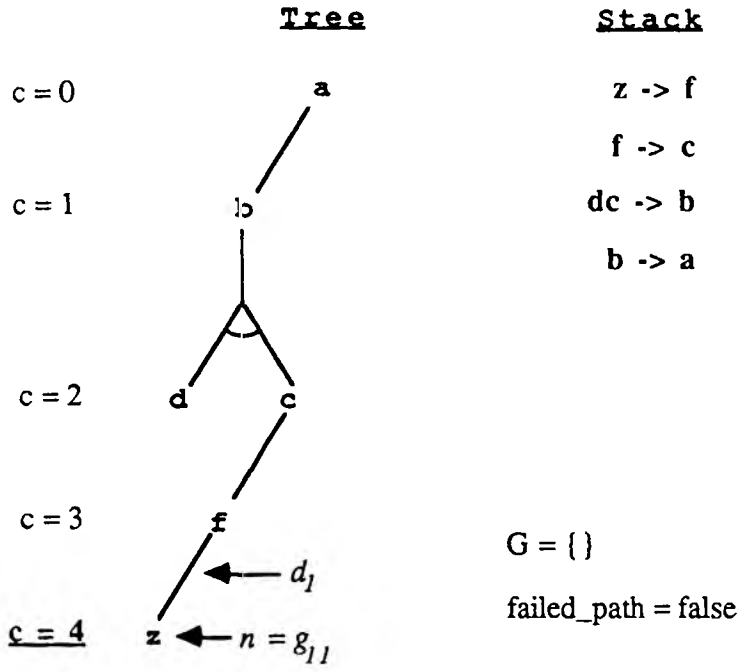

Fig. 8 Tree expanded from node f to node z by using $z \rightarrow f$.

Noting that c = 4 = s and $z \notin \{dq\}$, the failed_path flag is set to true in step 7. Since the search has failed, this branch of the search tree will be abandoned. The process moves to step 8. The attribute **q** has not been found in the tree so *n* is set to **f** the ancestor of **z**, and **c = 3**. Since the failed_path flag is true, $z \rightarrow f$ is popped from the stack. The tree and stack are shown in **Fig. 9**.

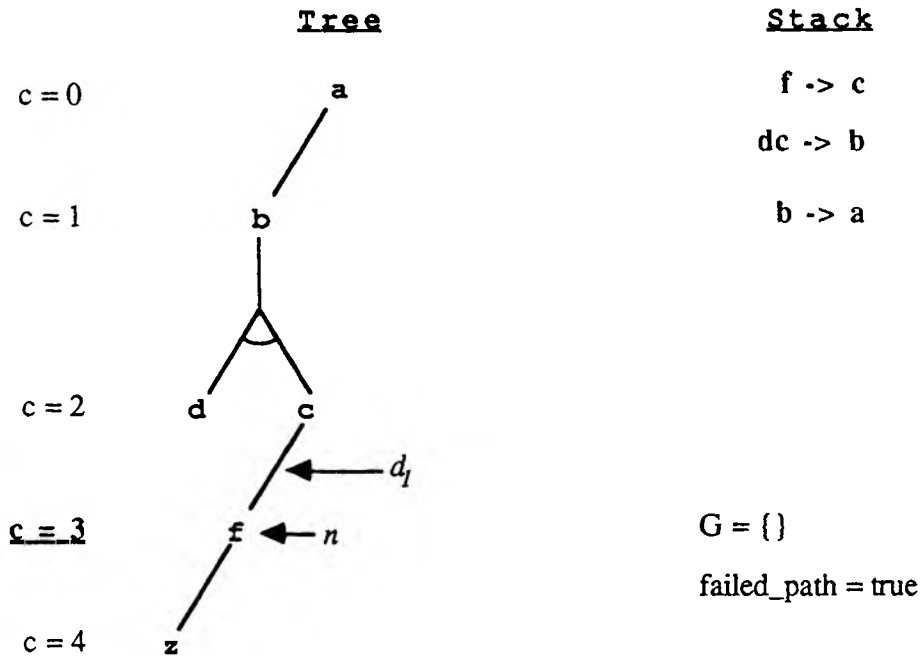

Fig. 9 Since z ∉ {dq} and the maximum tree depth is four, backtrack up the tree to node f and remove z → f from the stack.

As g_{12} does not exist and set G = { }, steps 8 and 9 are repeated. In this example, there is no opportunity to expand the width of the tree until n is equal to the root node **a**, hence the process continues repeating steps 8 and 9 popping the stack and moving n back up the tree. The 8 - 9 sequence of steps terminate when n = **a**, the root node. At this point the tree can be expanded, i.e. d_2 can be created. After reaching the root node, the tree and stack look as shown in **Fig. 10**.

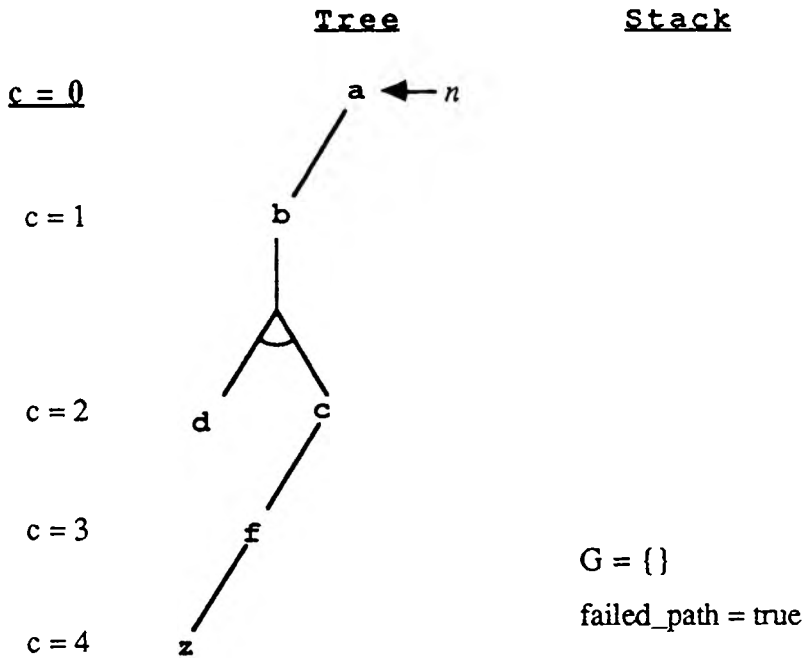

Fig. 10 Tree and stack after the DTA backtracks to node a.

Execution of step 10 leads to the repeating of steps 4 - 7, with n = **a**, and G = {**gf** → **a**}. A new branch d_2 is created from the root node and **gf** → **a** is pushed on to the stack. At the end of step 5, the tree and stack are as shown in **Fig. 11.**

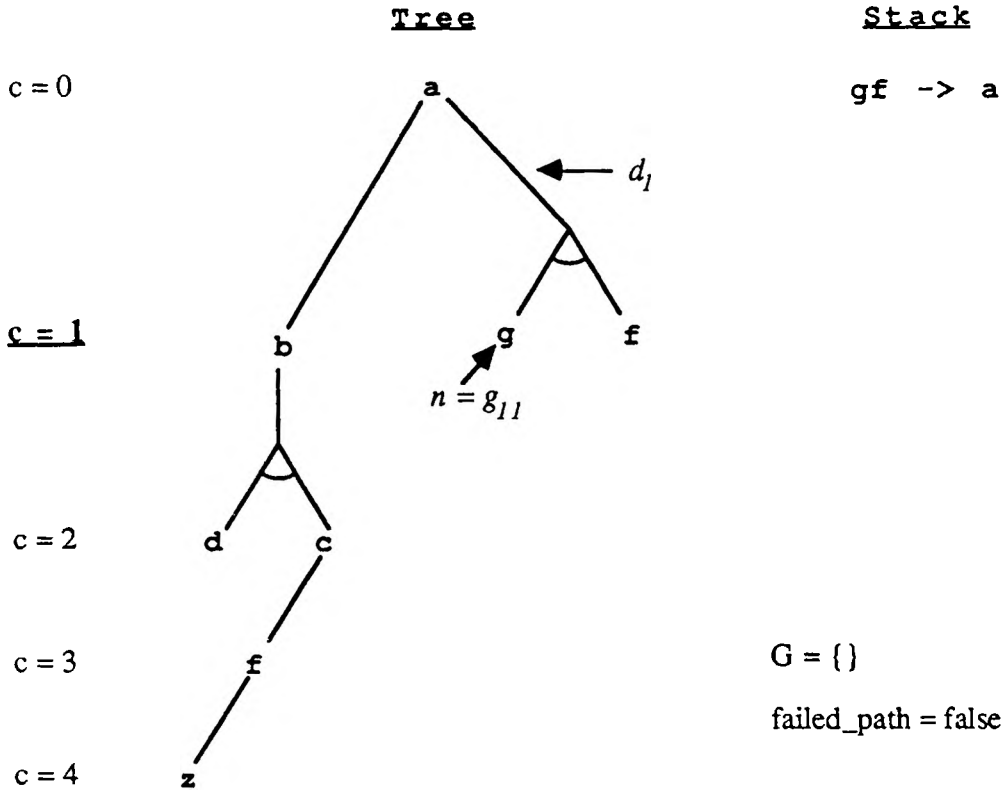

Fig. 11 Expansion of the tree from node a using gf → a.

Step 6 returns the process to step 4 to start expanding the depth of the tree from node g. With n = **g** and G = {**dr** → **g**}, branch d_1 is built from **dr** → **g** in step 5,

dr → **g** is pushed on the stack,

failed_path = false,

G = G - {**dr** → **g**} = { },

$n = g_{11}$ = **d**, and

c = 2.

The tree and stack now shown in **Fig. 12**.

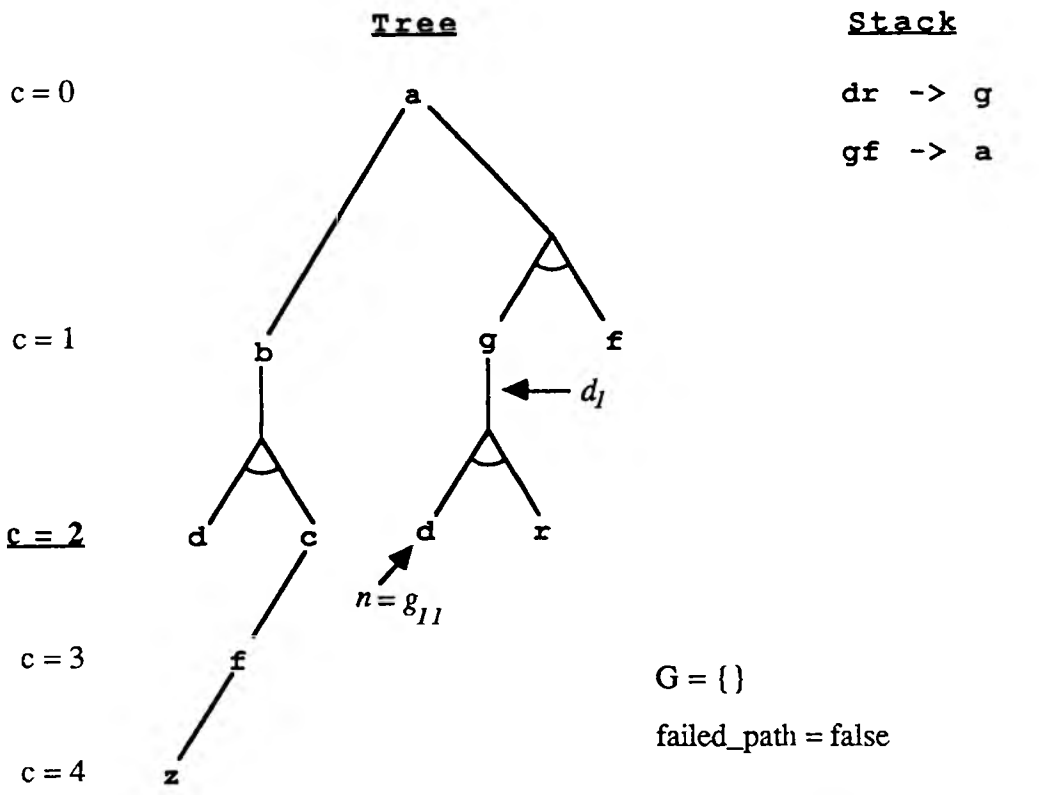

Fig. 12 Expansion of the tree from node g using dr → g.

As n = **d** and **d** ∈ {**dq**} the process moves to step 8 and starts developing the width of the tree at the current level 2. The attribute **q** has not been found so n is set to **g** which is the ancestor node of **d** and c is decremented to 1. The tree now looks as shown in **Fig. 13.**

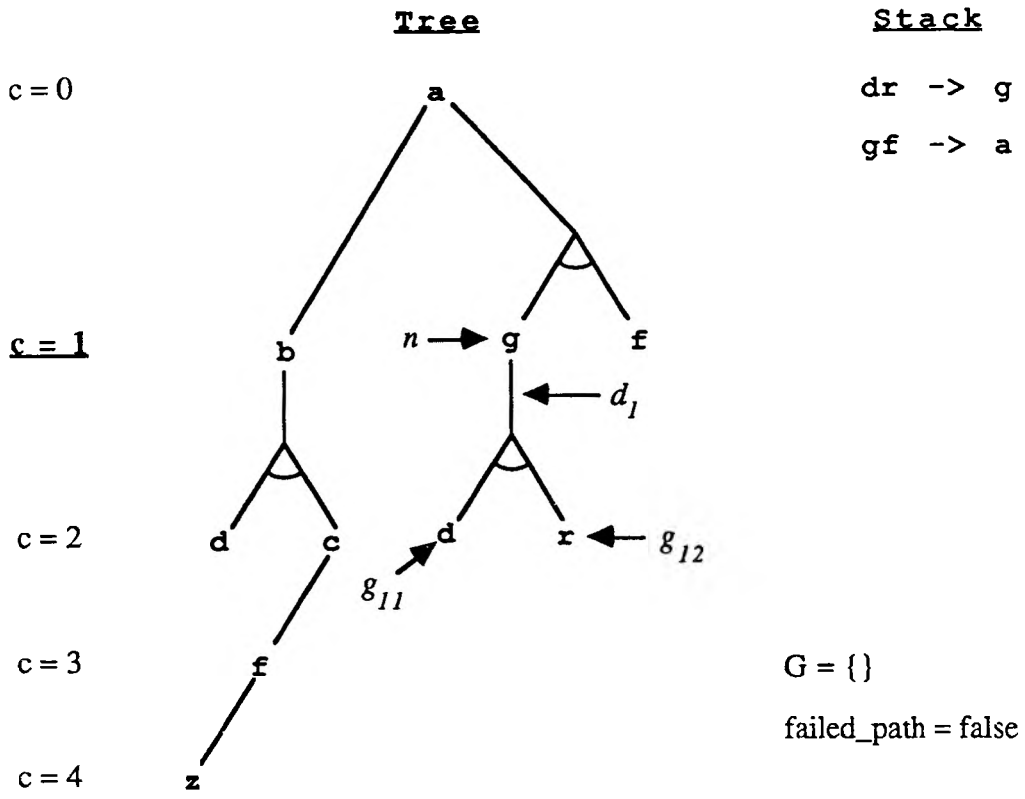

Fig. 13 DTA tree and stack prior to expansion at node r.

In step 9 since g_{12} exists, n is moved to g_{12} = **r** and c is incremented to 2.

Continued application of the DTA from the node n = **r** expands the tree as shown in **Fig. 14**. In this figure, execution of step 5 has just been completed.

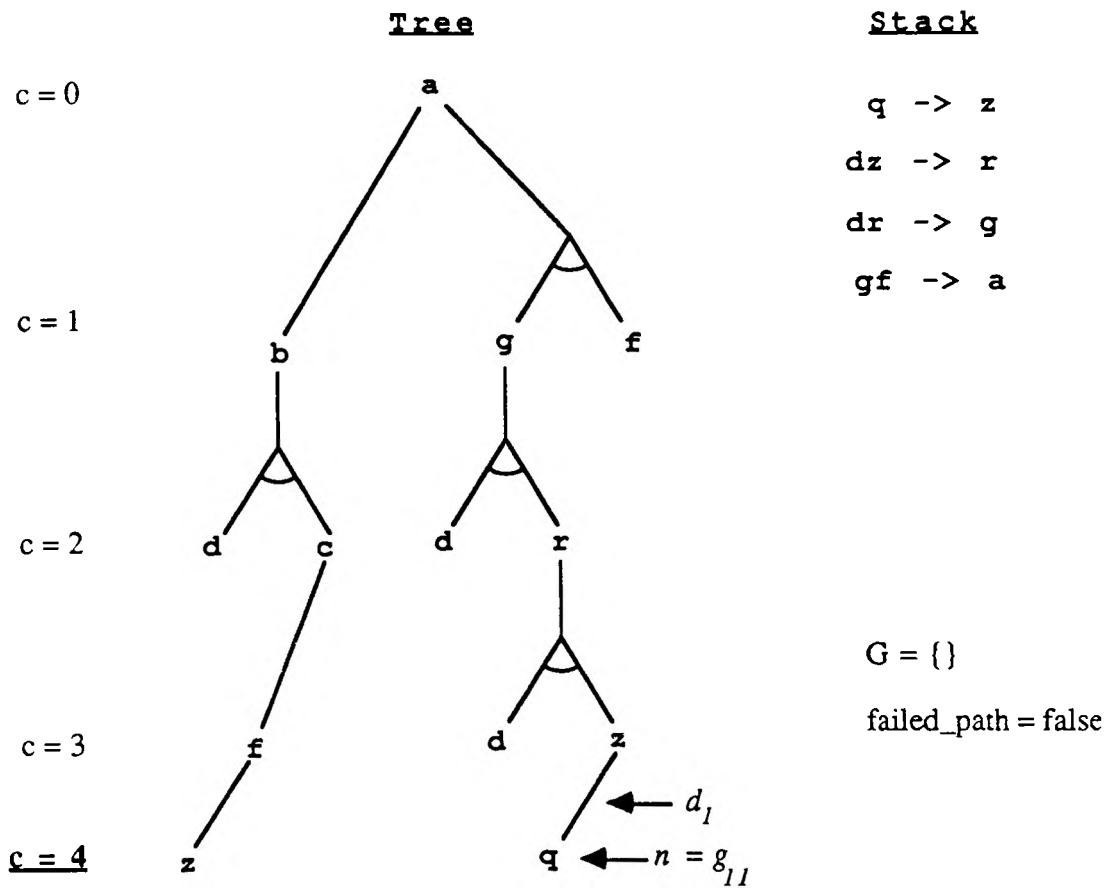

Fig. 14 Tree expansion stops at n = g_{11} since q ∈ {dq}.

Since **q** ∈ {dq} at step 6, the failed_path flag remains false at step 7 and the process advances to step 8. Even though **d** and **q** have both been found, the AND relation formed by **gf** → **a** has not been fully resolved. As $g_{i,j+1}$ does not exist for node **q**, **z**, or **r** and G = { } at each of these nodes the process repeats steps 8 and 9 until *n* is back at the root node where g_{12} = **f** still exists. At this point *n* is moved to node g_{12} = **f** and c = 1. Upon entry into step 10, the tree looks as shown in **Fig. 15**.

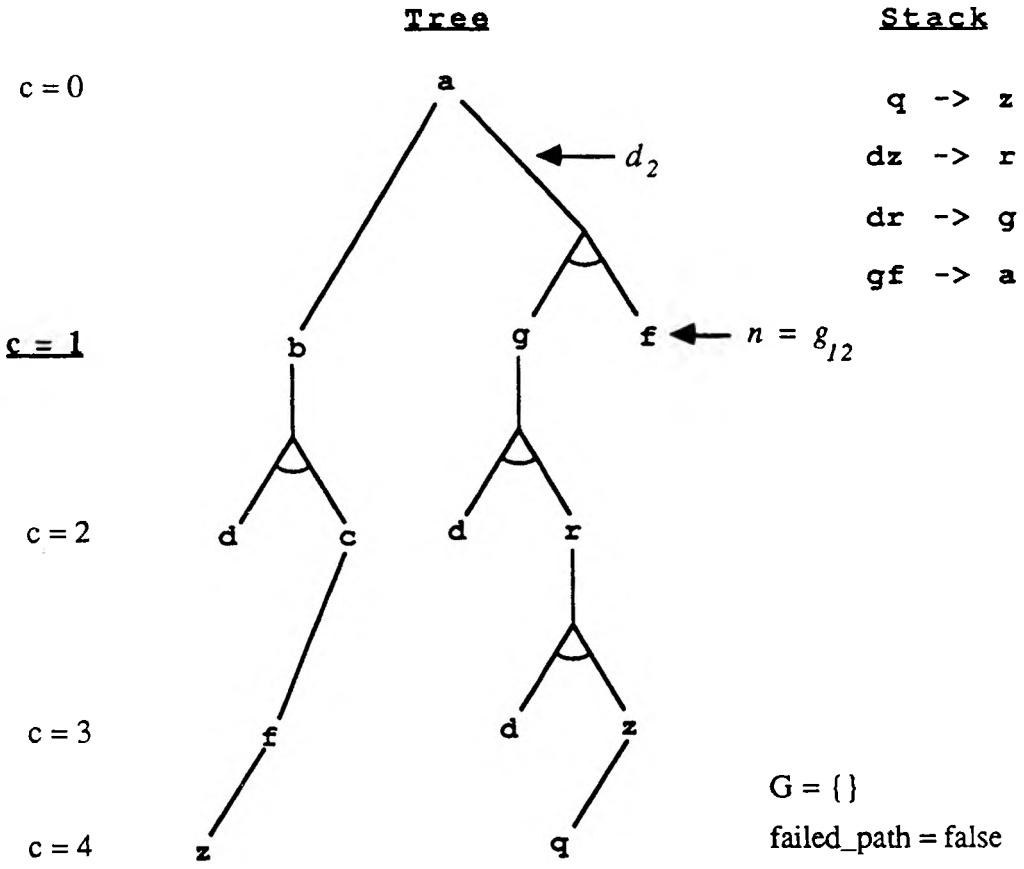

Fig. 15 DTA tree and stack prior to expansion at node f.

Step 10 returns the process to step 4 to start expanding the depth of the tree from n = f. The DTA continues to expand this portion of the tree until the situation shown in **FIG. 16** is reached. Execution of step 5 has just been completed.

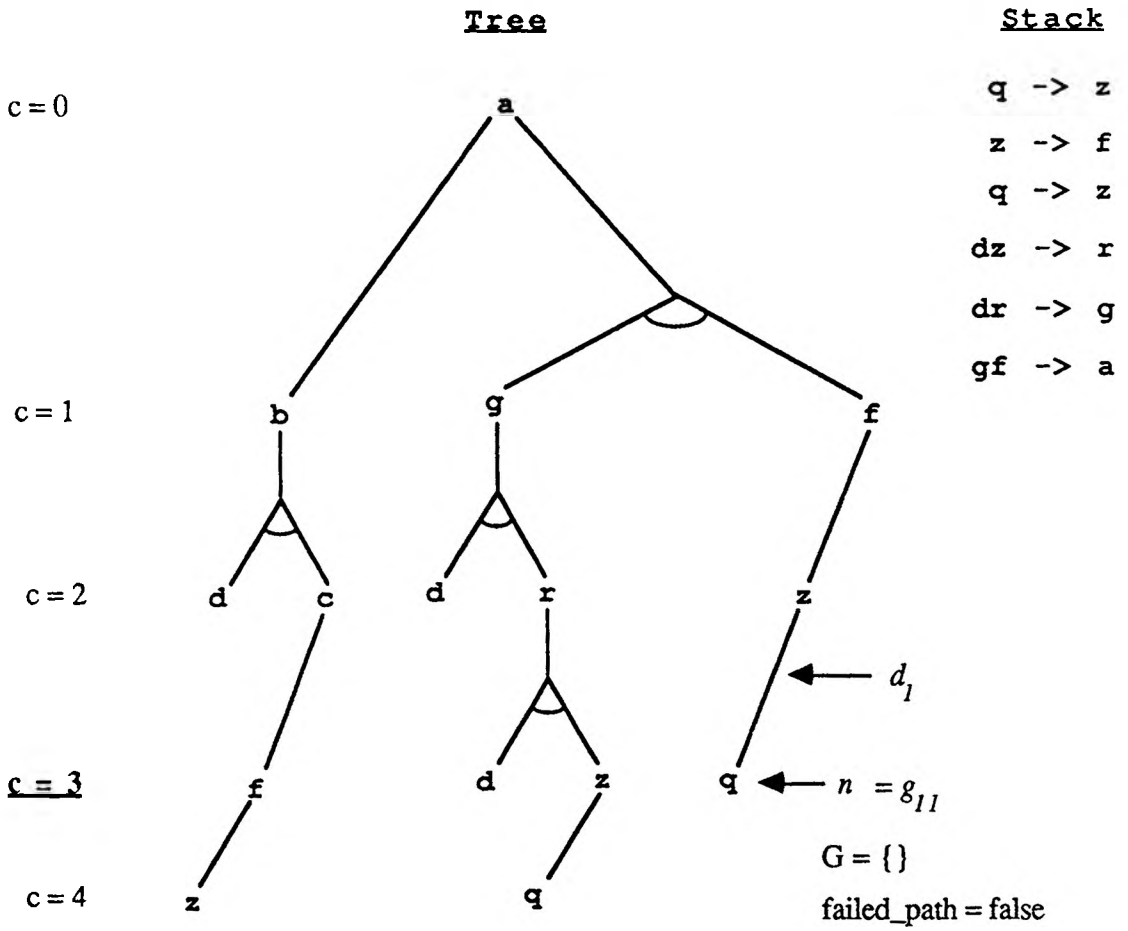

Fig. 16 Completed DTA tree and stack. The stack contains the desired trace.

At this point all attributes from goal set {**dq**} have been found and there are no partially resolved d_i remaining in the current tree. A deduction trace of **dq** → **a** is given by the contents of the stack and is represented by the set of functional dependencies;

$$\{\mathbf{q} \rightarrow \mathbf{z}, \mathbf{z} \rightarrow \mathbf{f}, \mathbf{dz} \rightarrow \mathbf{r}, \mathbf{dr} \rightarrow \mathbf{g}, \mathbf{gf} \rightarrow \mathbf{a}\} \subseteq F.$$

Features of the DTA

Each $X^{(i)}$ in the closure calculation relates, in reverse order, to a level in the deduction trace tree. The right hand side attribute, **r**, from $(L \rightarrow \mathbf{r}) \in F$ is used as the root node of the tree for the deduction trace and first shows up in $X^{(s)}$ in the calculation of $C_F(L)$. Some

of the elements of L will appear in the search tree before level s is reached, however since s is the smallest value for which $r \in X^{(s)}$, at least one of the attributes will require a search at level s. On the other hand, the deduction trace can be solved with a maximum tree depth of s. As in the above example for **dq** $\rightarrow$ **a**, since **a** $\in X^{(4)}$, the maximum depth of the tree required for a trace of **dq** $\rightarrow$ **a** is s = 4.

For a given minimal set of functional dependencies, F, an upper bound of the number of nodes which will be evaluated in constructing the AND/OR tree can be calculated. Let

$$R = \{ \mathbf{a} \mid (L \rightarrow \mathbf{a}) \in F \text{ and } \mathbf{a} \in U \}.$$

Since the value of a given **a** may appear in more than one right hand side, it is highly likely that |R| < |F|. For each **a** $\in$ R, let

$$S_{\mathbf{a}} = \{ L \mid (L \rightarrow \mathbf{a}) \in F \}.$$

Further, let

$$p = \max |S_{\mathbf{a}}| \text{ for } \mathbf{a} \in R.$$

Consider all $L_i \ni (L_i \rightarrow a_i) \in F$ for some i = 1,..|F|. Let

$$q = \max |L_i| \text{ for } i = 1,..|F|.$$

At a given node, the maximum number of d_i branches which can be derived is p. The maximum number of g_{ij} from any single d_i branch is q. Hence, the maximum number of g_{ij} nodes resulting from the expansion of any node **a** is pq. Further, an upper bound on the number of nodes in a combined AND/OR tree of depth s is:

$$\text{Upper bound} = \sum_{i=1}^{s} (pq)^i = \begin{cases} \dfrac{pq(1 - (pq)^s)}{1 - pq} & \text{for } pq \neq 1 \\ s & \text{for } pq = 1 \end{cases}$$

To illustrate the calculation, again consider the example problem where,

$R = \{\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{f},\mathbf{g},\mathbf{r},\mathbf{z}\}$,
$|R| = 7 < |F| = 8$,
$S_{\mathbf{a}} = \{ \{\mathbf{b}\}, \{\mathbf{gf}\} \}$,
$S_{\mathbf{b}} = \{ \{\mathbf{dc}\} \}$, and
$S_{\mathbf{c}} = \{ \{\mathbf{f}\} \}$.

Since S_f, S_g, S_r, and S_z are all singleton sets, p = 2. Futher, it can be seen that the largest number of attributes in any single left hand side is 2, so q = 2. Since the closure routine requires a minimum of four iterations to determine that $\mathbf{a} \in C_F(\{dq\})$, then s = 4. Using these numbers, an upper bound on the number of nodes generated by the DTA for this example is:

$$\frac{pq(1 - (pq)^s)}{1 - pq} = \frac{1020}{3} = 340$$

Hence, a maximum of 340 nodes would have to be searched to solve this problem. The value is a liberal upper bound since the problem actually required that 15 nodes be searched.

The tree produced by the DTA may not be unique. For example, applying the DTA to the minimal set of functional dependencies,

$$F = \{\mathbf{b} \rightarrow \mathbf{a},\ \mathbf{gf} \rightarrow \mathbf{a},\ \mathbf{jt} \rightarrow \mathbf{a},\ \mathbf{dc} \rightarrow \mathbf{b},\ \mathbf{f} \rightarrow \mathbf{c},\ \mathbf{z} \rightarrow \mathbf{f},\ \mathbf{dr} \rightarrow \mathbf{g},\ \mathbf{dz} \rightarrow \mathbf{r},$$
$$\mathbf{q} \rightarrow \mathbf{z},\ \mathbf{d} \rightarrow \mathbf{j},\ \mathbf{e} \rightarrow \mathbf{t},\ \mathbf{h} \rightarrow \mathbf{e},\ \mathbf{q} \rightarrow \mathbf{h}\}$$

and letting

$$\mathbf{dq} \rightarrow \mathbf{a}$$

be a functional dependency whose deduction trace is to be determined by the DTA gives either the deduction trace,

$$\begin{aligned} \mathbf{q} &\rightarrow \mathbf{z} \\ \mathbf{z} &\rightarrow \mathbf{f} \\ \mathbf{dz} &\rightarrow \mathbf{r} \\ \mathbf{dr} &\rightarrow \mathbf{g} \\ \mathbf{gf} &\rightarrow \mathbf{a} \end{aligned}$$

or the deduction trace,

$$\begin{aligned} \mathbf{q} &\rightarrow \mathbf{h} \\ \mathbf{h} &\rightarrow \mathbf{e} \\ \mathbf{e} &\rightarrow \mathbf{t} \\ \mathbf{d} &\rightarrow \mathbf{j} \\ \mathbf{jt} &\rightarrow \mathbf{a}. \end{aligned}$$

In both cases, s = 4. The solution produced by the DTA depends on the order in which the AND/OR tree is constructed. Both solutions are correct.

The above example illustrates the fact that more than one solution may be found by searching a maximum of s levels deep. However, searching more than s levels may produce additional solutions. In the present example, searching five levels deep will yield the solution,

$$\begin{aligned} \mathbf{q} &\rightarrow \mathbf{z} \\ \mathbf{z} &\rightarrow \mathbf{f} \\ \mathbf{f} &\rightarrow \mathbf{c} \\ \mathbf{dc} &\rightarrow \mathbf{b} \\ \mathbf{b} &\rightarrow \mathbf{a}. \end{aligned}$$

If $L \rightarrow r$ can be deduced from F, then r first shows up at least once in $X^{(s)}$ of $C_F(L)$. The number of times that r can be deduced in $X^{(s)}$ directly relates to the number of solutions to the DTA within a tree of maximum level s. In the above example $X^{(3)} = \{\mathbf{d,q,z,j,h,c,f,r,g,t}\}$, in the calculation for $X^{(4)}$, $\mathbf{gf} \rightarrow \mathbf{a}$, and $\mathbf{ct} \rightarrow \mathbf{a}$ can both be resolved so the number of solutions within a tree of depth 4 would be 2.

Conclusions

This paper describes an algorithm that will produce a deduction trace of $L \rightarrow r$ over a minimal set of functional dependencies F provided $r \in C_F(L)$. The depth of the search tree depends on the number of calculations required to determine that $r \in C_F(L)$. This algorithm will show the first solution found in the search of the AND/OR tree. If more than one solution exists, the order in which the tree is built will determine the solution produced. Ordering the AND/OR tree so the d_i with the smallest number of g_{ij} is evaluated first may produce a more nearly optimal solution. It may be desirable to see all of the

possible solutions within a tree of level s. In order to do this, the algorithm would have to be modified to store the number of possible solutions as calculated by the closure routine. As long as more solutions exist, the algorithm can return to the root node after each solution and start searching the next d_i emanating from the root node.

With computer generated tools to remove redundancies and extraneous left hand side attributes from a list of functional dependencies, the process of creating relational databases with minimal sets of functional dependencies has been simplified. With the addition of the DTA to resolve dependencies not contained in a single relation, it will be easier for relational database users to determine how to retrieve information from the database.

References

1. W. Kent, "A Simple Guide to Five Normal Forms in Relational Database Theory," *Communications of the ACM,* Febuary 1983, pp. 120-125.

2. S. Ceri and G. Gottlob, "Normalization of Relations and Prolog," *Communications of the ACM*, June 1986, pp. 524-544.

3. Betty Salzberg, "Third Normal Form Made Easy," *ACM SIGMOD Record*, December, 1986, pp. 2-18.

4. W. W. Armstrong, "Dependency Structures of Data Base Relations," *Proc. 1974 IFIP Congress,* North Holland, Amsterdam, pp. 580-583.

5. Jeffery D. Ullman, *Principles of Database Systems*, 2nd ed., Computer Science Press, Rockville, Md., 1982.

6. Nils J.Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill Book Company, New York, 1971.

VITA

James Morris Richardson was born on July 11, 1945 in Benton Harbor, Michigan. He received his primary and secondary education in Benton Harbor, Michigan and Toole, Utah. He attended Austin Community College, in Austin, Texas. In May 1979 he received an Associate of Arts degree in Liberal Arts from Meramec Community College in Kirkwood, Missouri. He received a Bachelor of Science degree in Applied Mathematics from the University of Missouri-Saint Louis, Saint Louis, Missouri in December 1980. He has been enrolled in the Graduate School of the University of Missouri-Rolla since September 1981. He is currently employed by McDonnell Douglas Information Systems Group, Saint Louis, Missouri.