

M.Tech. (Computer Science) Dissertation Series

# **Processing of Path Queries on Trees and DAGs**

a dissertation submitted in partial fulfillment of the  
requirements for the M.Tech. (Computer Science)  
Degree of the Indian Statistical Institute

*By*

**Ch. Bhanu Teja**

under the supervision of

**Prof. Aditya Bagchi**  
**CSSC**

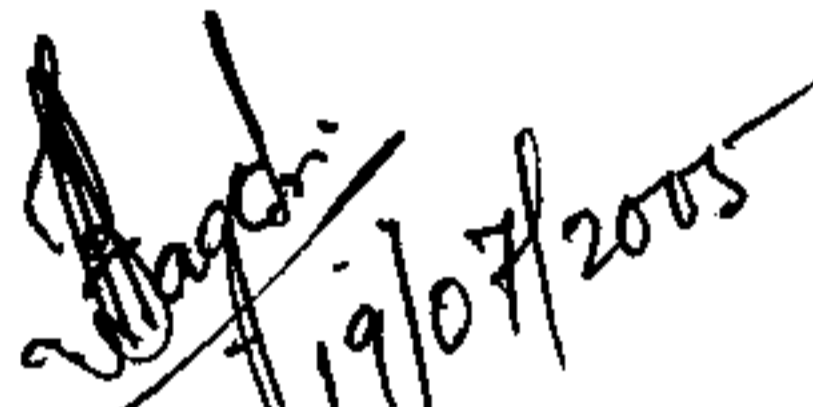


**INDIAN STATISTICAL INSTITUTE**  
203, Barrackpore Trunk Road  
Calcutta-700 035

## Certificate of Approval

This is to certify that this thesis titled "*Processing of path queries on trees and DAGs*" submitted by **Ch.Bhanu Teja** towards partial fulfillment of requirements for the degree of M.Tech. in Computer Science at Indian Statistical Institute, embodies the work done under my supervision.

The dissertation report may be accepted in partial fulfillment of the requirement for the M.Tech. (Computer Science) degree of Indian Statistical Institute, Kolkata, India.

  
(Supervisor)

(External Examiner)

## **ACKNOWLEDGEMENTS**

I take immense pleasure in thanking Prof. Aditya Bagchi (Professor, CSSC, Indian Statistical Institute) for providing me an interesting problem and guiding me through out the dissertation work. I admire his patience for hearing to my petty ideas and I am grateful to him for providing me invaluable suggestions. His pleasant and encouraging words have always kept my spirits up.

I would also like to thank my friends Vishnu Vardhan, Deepak Kumar, Lokesh Kumar and Shyam Sudhakar Chaturvedi for encouraging me through out the dissertation.

Kolkata, 11<sup>th</sup> July 2005.

Ch.Bhanu Teja

## **ABSTRACT**

Given a large graph, stored on disk, there is often need to perform a search over this graph. Such, a need could arise, for example, in the search component of a data-intensive expert system or to solve path problems in deductive database systems or simply, to search on the web-graph. In this dissertation, we present a novel graph compression and data structuring technique on the compressed graph and show that the search algorithms can use this data structure to prune the search space, thereby improving the efficiency. We advocate the use of labeling scheme proposed by H.V.Jagadish [1] combined with our data-structure, for modeling this type of graphs in-order to efficiently answer the queries such as ancestors, descendants, and least common ancestors which usually require costly transitive closure computations. We also provide algorithms for finding all common ancestors, all common descendants and all paths between any two given nodes.

# Contents

1. Introduction	1
2. Path Normalization	4
2.1. Generation of Normalized Set of Paths	5
2.1.1. Algorithm for construction of Normalized Paths	7
2.2. Construction of Node Buckets	13
2.3. Reconstruction Process	15
3. Query Processing for Directed Trees	18
3.1. Pre-processing	18
3.2. Algorithms for query processing	20
3.2.1. Reachability	21
3.2.2. Least Common Ancestor of node U and node V	21
4. Graph Compression	23
4.1. Algorithm for converting directed graph to DAG	24
5. Query Processing for DAGs	28
5.1. Pre-processing	28
5.2. Query Processing	35
5.2.1. Reachability	35
5.2.2. Find all ancestors of a given node X	35
5.2.3. Find all common ancestors of nodes X and Y	36
5.2.4. Find all descendants of a given node X	36
5.2.5. Find all common descendants of nodes X and Y	37
5.2.6. Find Least Common Ancestors for any given pair of nodes X and Y	37
5.2.7. Find all the paths between the nodes U and V	39
6. Conclusion and Future Work	44
 Bibliography	 45

# Chapter 1

## Introduction

Many present day applications like, modeling of biological pathways, social networks, and digital libraries use graph based data structure for representing their data. So, the research community is also showing increasing interest towards developing methods for storage and processing of large graphs. Surprisingly, a large portion of such modeling efforts involve directed acyclic graphs (DAG). For example, terminology graphs of the Gene Ontology Consortium [13] induce DAG-structured sub graphs through *is-a* and *part of* edges. One of the largest DAG-structured terminology graphs is the classification system of the US Patent and Trademark Office [14], which has several hundred thousand categories to date. Another very important area of modern day application is the formation of Digital Library consortium that uses DAG as the basic data structure. So, special study needs to be made to develop efficient mechanism for processing queries on DAG structure. A recent work [8] has addressed the problem of pattern matching for DAG-structured data which exist in many scientific and ontological databases. By utilizing auxiliary data structures to hold partial solutions and exploiting the temporal properties in node processing, it has generalized the original PathStack and TwigStack algorithms to be applied to DAGs. However, his research effort basically extends the seminal work of Holistic twig joins [9] primarily meant for XML pattern matching. Some applications like social network, digital library consortium etc. are primarily interested in path based searching and efficient path search is more important than twig pattern matching.

The present work is the outcome of an effort to model social network applications [2] in order to handle a large directed graph representing a social community. The graphs in social network are directed graphs. So, by graph we mean directed graphs unless otherwise specified. A graph compression technique has been used to fuse each strongly connected component to a hyper-node, turning a graph to a DAG. So that, further processing can be done on the DAG.

Since the application area under study mainly demands path-based queries, two different approaches may be taken to handle them:

- Either by creating paths from the edges against a query, or
- By generating all paths *a priori* and then selectively retrieving them against a query.

However, a real life graph would involve hundreds or even thousands of nodes. A social network application usually involves a part of the web graph as data. In a DAG of hundreds of nodes and edges, computation of paths against queries will make query processing very slow. So, the first approach will be time prohibitive. However, storing all the pre-computed simple paths of different lengths, indexing them and then accessing them against queries may not make the query processing more efficient. This would also need too much of space. So, a trade-off is necessary, where a few paths and sub-paths will be pre-computed and stored and all the simple paths of the DAG should be computable from them. This process has been called as **Path Normalization** to draw similarity with the corresponding effort taken in the well-known relational data model. The present work describes the normalization process in detail and considers it to be minimal and complete.

Here, minimality signifies that in the normalized path set, an edge of the original graph will appear only once. On the other hand, completeness ensures that through appropriate procedure, all simple paths of the original graph can be reconstructed from the normalized pathset without generating any extra path.

The main advantages derived from the present work are:

- a) Depending on the query and the corresponding search, existing paths in the normalized pathset may suffice to answer the query. So, no new paths are generated during query processing.
- b) In case new paths need to be generated, only the minimal number of paths necessary over the existing normalized pathset will be generated.

The most common queries raised by the users of social networks have been collected. The efficacy of the path normalization process has been studied by developing algorithms to handle these queries for directed trees and DAGs. The queries are:

- a) Reachability queries.
- b) Finding all ancestors of a node. During this process of finding ancestors, the structural information of ancestors can also be obtained.
- c) Given any two nodes, finding all common ancestors.
- d) Given any two nodes, finding the least common ancestor.
- e) Finding all descendants of a node.
- f) Given any two nodes, finding all common descendants.
- g) Generation of all paths between two nodes.



## Chapter 2

### Path Normalization

The network based real life applications considered for our study are represented as digraphs. After the fusion of the *Strongly Connected Components* as hyper-nodes, the augmented network is represented as a DAG. The DAG can be stored as adjacency lists. The *paths* of the graph are computed from the adjacency lists. A path is a sequence of nodes. The applications under consideration mainly involve path searches, such as

*“Find all paths passing through nodes  $n_a$  and  $n_b$ ”*

One simple way for such path determination is to generate all *complete paths*. A complete path is defined as a node sequence from a *Source* (node whose in-degree is 0) to a *Sink* (node whose out-degree is 0). These complete paths may be stored and indexed with proper ids. Now, find the paths containing the nodes  $n_a$  and  $n_b$ . If we achieve this by storing all complete paths, the followings things may happen, if two complete paths are considered,

- They may either be isolated or
- The two paths have a common sub-path that can be obtained by the intersection of the corresponding node sequences.

Storing all complete paths will lead to many redundant node sequences corresponding to the common sub-paths. Normalization process should ensure that only non-redundant paths and sub-paths are generated and stored.

A special kind of node called *anchor node* plays a very important role in the normalization process. An anchor node is a non-source node with out-degree greater than 1 or a non-sink node with in-degree greater than 1. The anchor nodes and their corresponding in-degrees or out-degrees determine the cardinality of the set of non-redundant paths and sub-paths generated during normalization. In the next section we discuss how to generate normalized set of paths.

## 2.1 Generation of Normalized Set of Paths

First, let us consider the problem informally. If any three nodes of a DAG are considered, they can be connected in only three ways. We call them *Chain*, *Cap* and *Cup*.

The *Chain structure*, as shown in Fig 2.1, is a sequence of three nodes where two consecutive nodes are directly connected. So in this structure, the DFS tree generated from node  $n_2$  in the graph is inevitably a part of the DFS tree generated from  $n_1$ . Similarly, the DFS tree generated from node  $n_3$  in the graph is a part of the DFS tree generated from  $n_2$  and hence of  $n_1$  as well. So, the DFS tree from node  $n_1$  will generate all the possible paths from  $n_1$  and will cover all the paths starting from  $n_2$  and  $n_3$ .

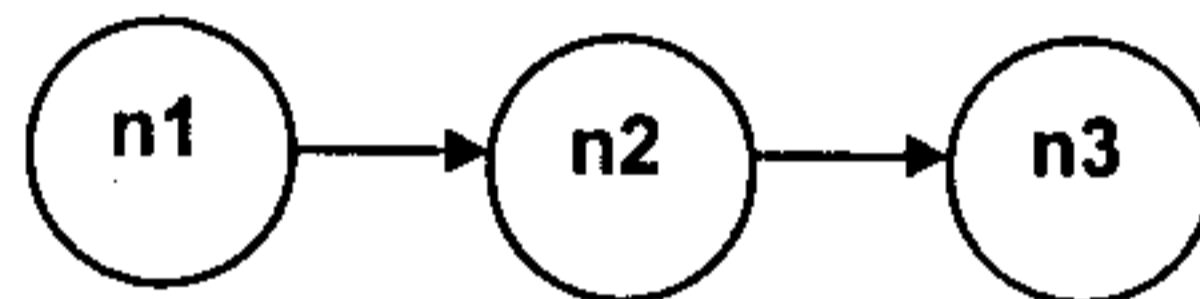


Fig 2.1 Chain Structure

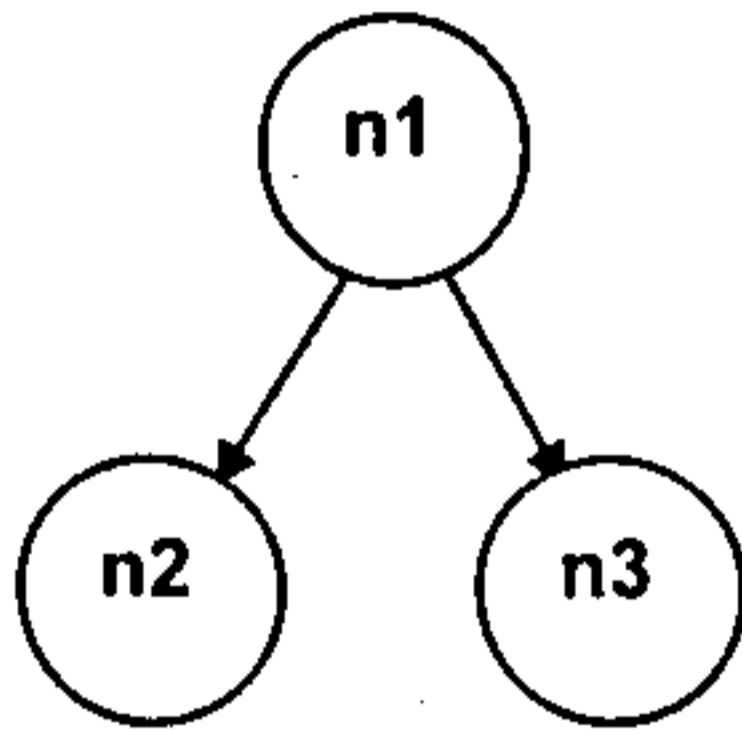


Fig 2.2 Cap Structure

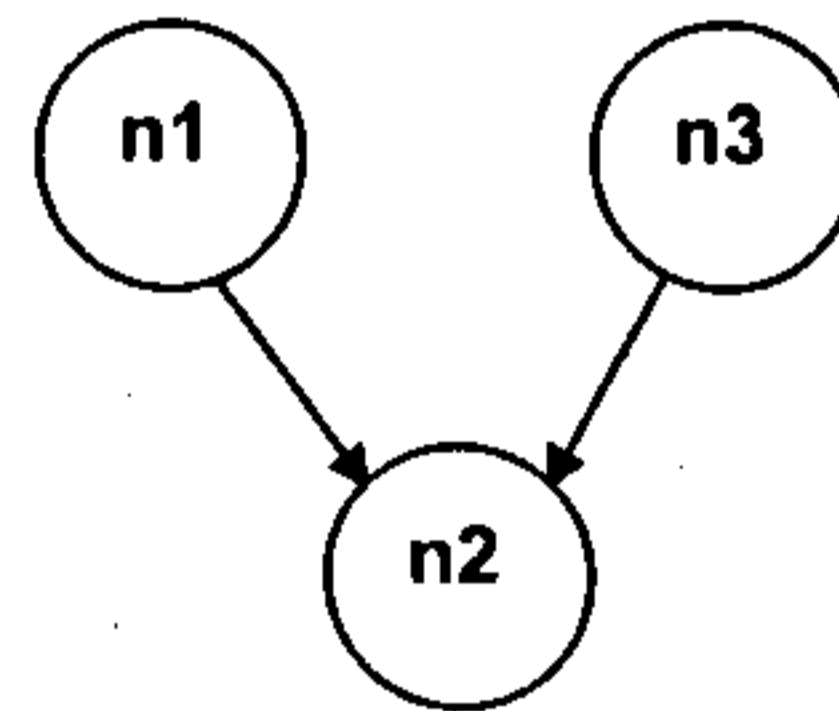


Fig 2.3 Cup Structure

The *Cap structure*, of Fig 2.2, shows two nodes  $n_2$  and  $n_3$ , linked to a root node  $n_1$  giving it a cap-like structure. Hence, the DFS tree generated from  $n_1$  will totally cover the DFS trees generated from both  $n_2$  and  $n_3$ . So, the paths generated from  $n_2$  and  $n_3$  are all contained in the paths generated from  $n_1$ .

The *Cup structure*, as shown in Fig 2.3, exhibits two nodes  $n_1$  and  $n_3$  linked to a node  $n_2$  giving it a cup-like appearance. Here, the DFS tree under  $n_2$ , is a part of the DFS tree starting from either  $n_1$  or  $n_3$  because both covers node  $n_2$ . So if the paths are generated from both  $n_1$  and  $n_3$ , paths from  $n_2$  will appear twice. So, in order to avoid such redundancy, if paths from  $n_1$  are generated then from  $n_3$  only the sub-paths between  $n_3$  and  $n_2$  need to be generated.

Once again according to the earlier discussion, in the Cup structure, as shown in Fig 2.3, if both node  $n_1$  and node  $n_3$  are natural roots, all the complete paths from both  $n_1$  and  $n_3$  will be generated and stored. However, all the paths from node  $n_2$  will be sub-paths for both sets of complete paths. Path normalization process should avoid this redundant storage of sub-paths.

So in the path normalization process, if the path generation starts from node  $n_1$ , all the complete paths from  $n_1$  are generated and stored. In case of node  $n_3$ , only the edge  $(n_3, n_2)$  is stored, since the paths from  $n_2$  have already been considered as sub-paths of the complete paths generated from node  $n_1$ .

Similarly, if the path generation starts from node  $n_3$ , all the complete paths from  $n_3$  are generated and stored. Here, in case of node  $n_1$ , only the edge  $(n_1, n_2)$  is stored, since the paths from  $n_2$  have already been considered as sub-paths of the complete paths generated from node  $n_3$ .

So depending on the starting node, the normalized set of paths in above two cases are different.

Informally speaking,

- The normalization process generates and stores non-redundant set of paths only, and
- The resultant set of paths obtained out of the normalization process is not unique.

So, path normalization results in a set of paths for the graph that contains complete paths

(node sequence from a Source to a Sink) as well as sub-paths shared by two or more complete paths. Using the normalized set of paths, all the complete paths of the graph can be generated. Reconstruction process of generating all complete paths from these normalized paths has been proposed.

### 2.1.1 Algorithm for construction of Normalized Paths

The standard DFS algorithm controls the search process by assigning appropriate colors to the nodes. The different colors are,

**WHITE:** A node that has not been visited yet. Initially all nodes are white.

**GRAY:** A node that has been traversed at least once but all paths out of it have not been traversed.

**BLACK:** A node is made black when all paths out of it are traversed.

In the algorithm,  $V(G)$  denotes the set containing all the nodes of the graph  $G$ ,  $\pi(u)$  denotes the predecessor of node  $u$  and  $Adj(u)$  denotes the set containing the adjacent nodes of  $u$ . **DFS** function has the Graph as input whereas the **DFS\_VISIT** function has a node as the input and traverses the Graph recursively.

The standard DFS algorithm is modified to generate the normalized set of paths. The modified algorithm is given below:

```
DFS (G)
begin
  for each vertex  $u \in V(G)$ 
  do begin
    color  $[u] \leftarrow$  WHITE
     $\pi(u) \leftarrow$  NIL
  end;
  for each vertex  $v \in V(G)$ 
    if in-degree( $v$ )=0 then DFS_VISIT( $v$ )
end;
```

```

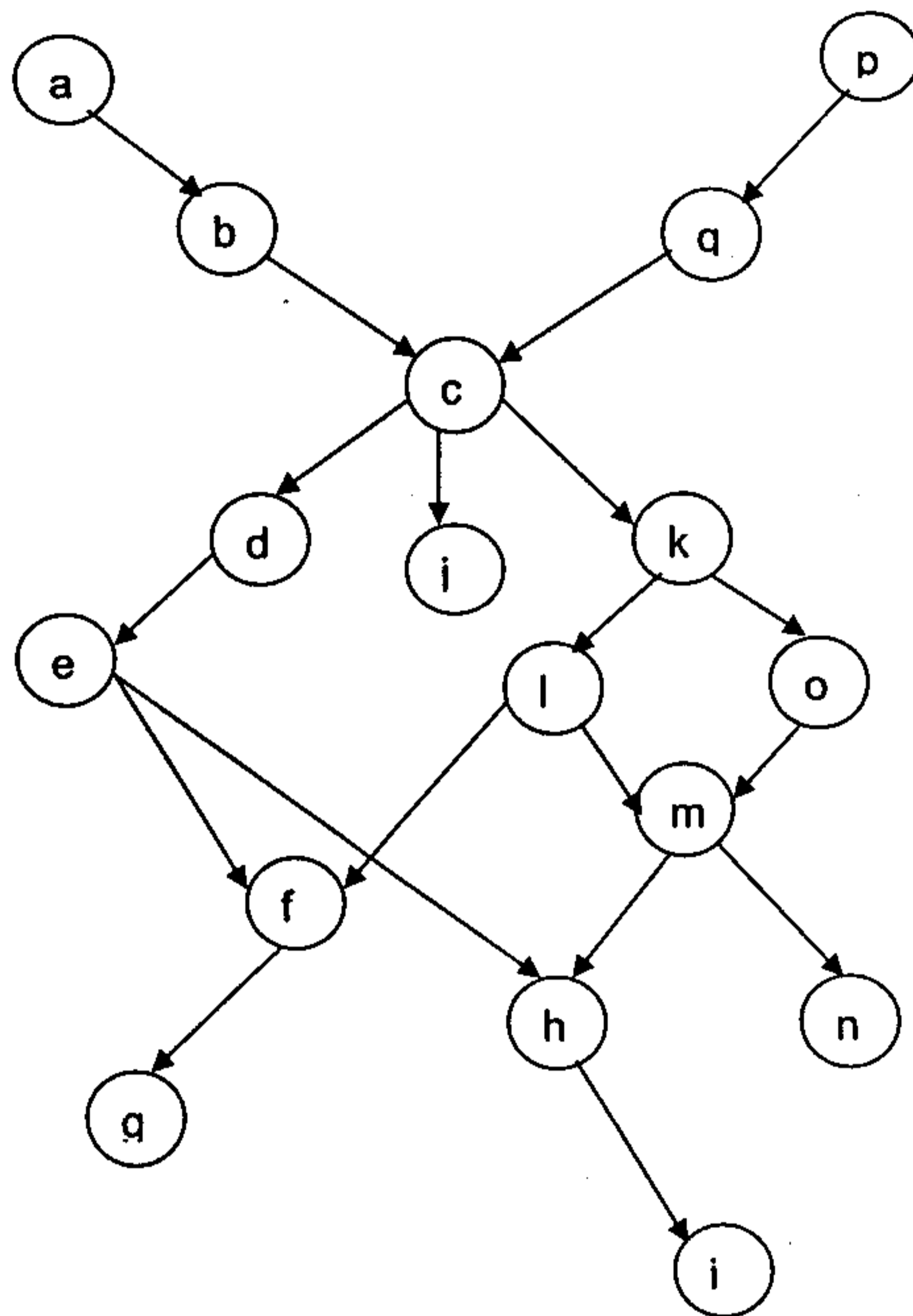
DFS_VISIT (v)
begin
  if Adj[v] ≠  $\phi$  and color[v] ≠ BLACK
  then begin
    color[v] = GRAY
    for each vertex x ∈ Adj[v] do
      if color[x] = WHITE or GRAY
      then begin
        color[x] = GRAY
        path = path U (v,x)
        DFS_VISIT(x)
        end;

      else begin
        path = path U (v,x)
        store path with new-id
        path =  $\phi$ 
        end;
      end;
    end;
  else begin
    color[v] = BLACK
    store path with new-id
    path =  $\phi$ 
    end;
  color[v] = BLACK
end;

```

The modified algorithm recursively generates the normalized set of paths from each Source (i.e. node with in-degree 0). The salient features of the algorithm are:

- In the normalized set of paths generated from this algorithm, the starting node of any member path (complete path or a sub-path) is either a Source or an Anchor node that is already a member of another path but still has successors.
- Any member path in the normalized set of paths either terminates at a Sink or at a node which has already been colored BLACK (i.e. all its successor nodes have already been visited).
- If path generation is done for Source nodes only, standard DFS algorithm will generate only the complete paths with all redundant edges repeated. The modified algorithm will consider an edge only once in the normalized set of paths. Both the algorithms are applied on the graph shown below.



**Fig 2.4** Example directed acyclic graph

Following the modified algorithm, the normalized set of paths is:

P1: [1, 1] {a, b, c, d, e, f, g, h}

P2: [0, 1] {e, h, i}

P3: [0, 1] {c, j}

P4: [0, 0] {c, k, l, f}

P5: [0, 0] {l, m, h}

P6: [0, 1] {m, n}

P7: [0, 0] {k, o, m}

P8: [1, 0] {p, q, c}

However, the query processing may demand the reconstruction of all the complete paths from the normalized set of paths. For the purpose of such reconstruction, some additional information is associated with each normalized path. The normalized paths are categorized in 4 types as shown in the square bracket associated with each path.

- If a normalized path starts from a source node and ends in a sink node, it is a [1, 1] path.
- If a normalized path starts from a source node but ends in an anchor node, it is a [1, 0] path.
- If a normalized path starts from an anchor node and ends in a sink node, it is a [0, 1] path.
- If a normalized path starts from an anchor node and also ends in an anchor node, it is a [0, 0] path.

The reconstruction process is equivalent to the join process available in the well-known relational system.

Considering the way we have generated the normalized paths, the normalized paths for a DAG are not unique.



For example,

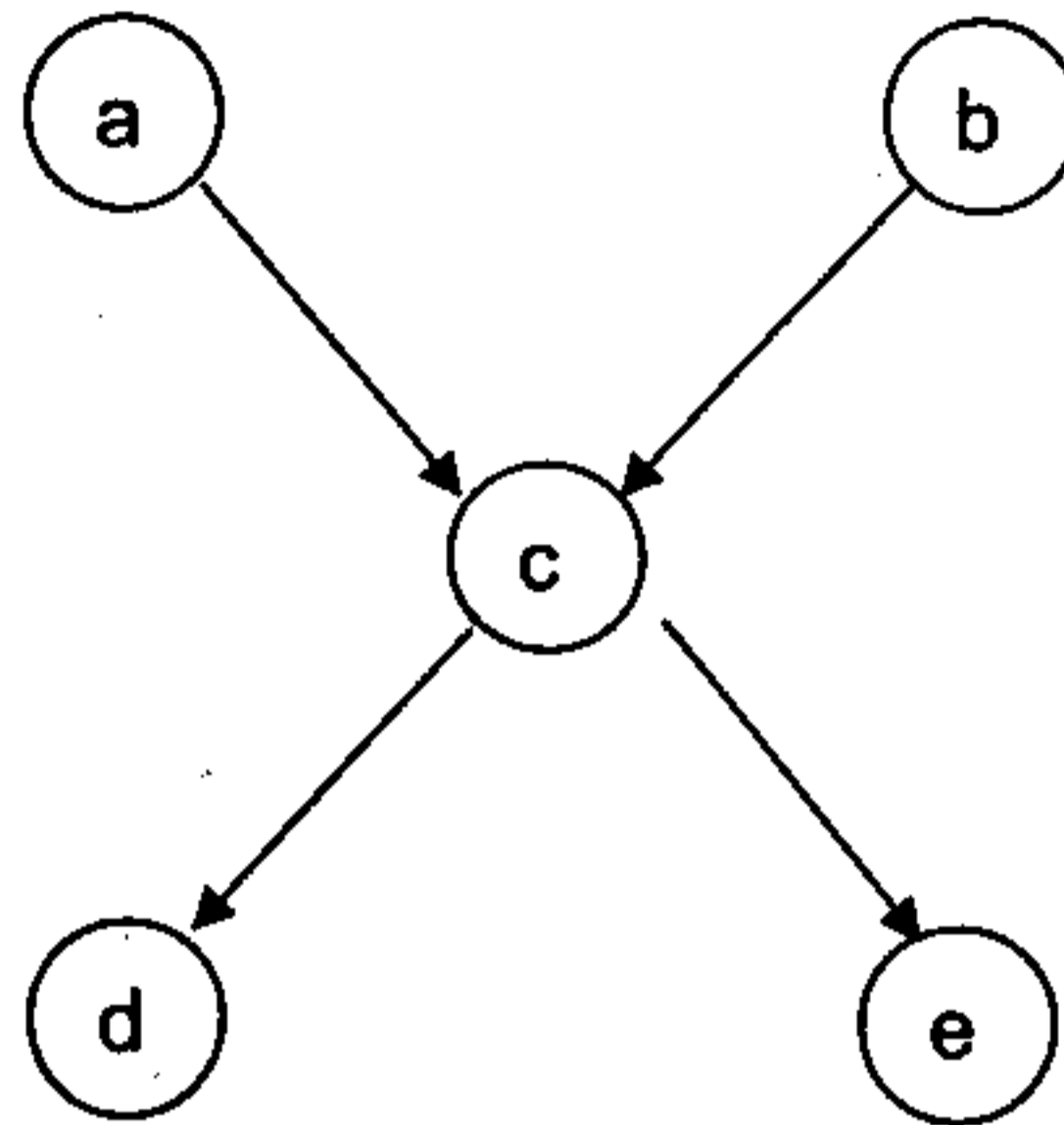


Fig 2.5 Example DAG

**Normalized path set 1:**

P1: {a, c, d}

P2: {c, e}

P3: {b, c}

**Normalized path set 2:**

P1: {a, c, e}

P2: {c, d}

P3: {b, c}

In this DAG Fig 2.5, we have presented two normalized set of paths, precisely there are 4 such paths.

From the above discussion of cup, cap and chain structures it is evident that the number of paths containing a source node ( $R_i$ ) is equal to its out-degree, for a sink node ( $L_i$ ) it is equal to its in-degree and for any other node ( $N_i$ ) it is equal to in-degree ( $N_i$ ) + out-degree ( $N_i$ ) - 1. Since, an intermediate node remains as an intermediate node in one path, it will appear as an end node in in-degree ( $N_i$ ) - 1 paths and as a start node in out-degree ( $N_i$ ) - 1 paths. So, the number of nodes we are storing is constant for a given DAG. Also, for any graph  $G = (V, E)$  the number of edges  $E$  is constant.

A path of  $n$  edges constitutes a sequence of  $n + 1$  nodes. So, for storing a path we need to store the number of nodes one more than the number of edges in the path. Thus, the

number of nodes stored is *equal to* number of paths + number of edges. Since, the number of nodes stored and the number of edges for a DAG is constant, the number of paths that are generated is also constant for a given DAG.

Construction of these normalized paths is done on the basis of occurrence of these paths in the path queries. Usually, it is done by the database designer.

One of the properties of the standard normalization process present in the well-known relational system is the redundancy avoidance. When all the features present in a problem domain are distributed among different tables, normalization algorithm tries to ensure that same feature is not repeated in different tables. However, such process cannot ensure complete avoidance of repetition.

**Lemma 1** Path normalization process avoids redundancy completely.

*Proof:* Redundancy avoidance will be considered complete only if no edge is repeated in the normalized set of paths. The path normalization process or the modified DFS algorithm executes a recursive function DFS\_VISIT that terminates either on a Sink node or if the visited node has the color BLACK. Once again, as soon as all the descendants of a node are visited or if a node does not have any descendant, its color is made BLACK. So, the recursive function DFS\_VISIT never accesses any node beyond a BLACK node. So, a node once made BLACK (i.e. its descendants are already visited), edges beyond it will never be visited twice. Recursively, the DFS\_VISIT function turns the color of the nodes from a Sink to a Source as BLACK keeping no scope of visiting the same edge more than once and subsequently to add them to a path and to store them. Hence the redundancy avoidance is complete.

## **2.2 Construction of Node Buckets**

In processing a query, it is necessary to retrieve a set of normalized paths. But since the graph considered is large, the number of these paths will also be high. The retrieval of these paths and performing union or join operations on them, while processing the query, will be costly in both space and time. So, to find and retrieve optimal number of paths

needed according to the query, the concept of node buckets was introduced.

**Definition:** A node-bucket represents a container of information which relates the node with the set of paths. It basically contains the set of paths containing the node and its related information. Generation of a path set simultaneously adds the identity of the path and related information to the node bucket.

Node bucket is simply a set of paths which when joined in different ways gives rise to a number of simple paths or part of a simple path from that node. So, querying against the nodes leads to the union or join operation on the contents of the node buckets to formulate the paths.

The information contained in each member of the node bucket is the path identity, originating node and the terminal node of the path. This information is used in the reconstruction of the simple paths from the node mentioned in the query.

With reference to the graph in Fig 2.4, the node buckets are:

a: [0, 1]	P1(a, g)
b: [1, 1]	P1(a, g)
c: [2, 3]	P1(a, g) P3(c, j) P4(c, f) P5(p, c)
d: [1, 1]	P1(a, g)
e: [1, 2]	P1(a, g) P2(e, i)
f: [2, 1]	P1(a, g) P4(c, f)
g: [1, 0]	P1(a, g)
h: [2, 1]	P2(e, i) P5(l, h)
i: [1, 0]	P2(e, i)
j: [1, 0]	P3(c, j)
k: [1, 2]	P4(c, f) P7(k, m)
l: [1, 2]	P4(c, f) P5(l, h)
m: [2, 2]	P5(l, h) P6(m, n) P7(k, m)
n: [1, 0]	P6(m, n)
o: [1, 1]	P7(k, m)
p: [0, 1]	P8(p, c)
q: [1, 1]	P8(p, c)

- the information in the square brackets after each node represents the in-degree and out-degree of that node, and the one in parentheses denotes the originating node and the destination node of the path.

### **2.3 Reconstruction Process**

The normalized set of paths has to be stored and indexed in such a way that the retrieval and reconstruction of original paths during query processing is efficient. However, to show the completeness and to generate all paths between any two given nodes, reconstruction of paths is needed. The algorithm presented below gives a way to generate all simple paths from the normalized paths and the corresponding node buckets.

## Algorithm for Reconstruction process

1. L is a global boolean variable that corresponds to the Leaf or not and t is initialized to 0 and sp is a pointer pointing to the elements in the respective node bucket.

2. Procedure RECONSTRUCT ( node )

```

1)   begin
2)   if L = 0 then
3)   sp:= bucket[node].start;
                                           ..... (i)
4)   else
5)   sp:=bucket[node].start->next;

6)   repeat
7)   get the pathset pointed by sp in the
      node buckets;
8)   for each node w E pathset do begin
9)   if node = w then break;
10)  end;

11)  for each node w E pathset do begin
12)  PUSH ( w,stack)
13)  end;
                                           ..... (ii)

14)  update the value of L according to the pathset;
15)  if L = 1 then begin
16)  Display the stack;
17)  end
18)  else begin
19)  get the last node of the pathset;
20)  w = POP(stack);
21)  RECONSTRUCT(last node);
22)  end;
                                           ..... (iii)

23)  repeat
24)  w:= POP(stack);
25)  if w not equal to node && bucket[w].outdegree > 1 then
26)  RECONSTRUCT(w);
27)  until num not equal to node;
                                           ..... (iv)

28)  while not end of contents of bucket[node] do begin
29)  sp:=sp->next;
30)  if sp not equal to NULL then
31)  if sp(origin) = node then break;
32)  end
32)  until sp is equal to NULL;
                                           ..... (v)

33)  end;

```

- if  $L = 0$  it is evident that the last node has in-degree  $> 1$  and node has been traversed before. So for the generation of all the paths from the last node, it has to start from the very first element in the respective node bucket pertaining to the last node, whereas in case of  $L=1$ , it is not needed.
- lines 7-13 it pushes the elements from the variable node till the end of the path set in the stack.
- lines 14-22, the  $L$  value is updated. if  $L = 1$  then display the stack note that the stack contains a complete path, else revisit the Reconstruct procedure with the last node obtained from the pathset as the complete path is yet to be obtained.
- lines 23-27, renders the pop out mechanism until a node is found that is not equal to the node value in the RECONSTRUCT procedure as well as have an out-degree  $> 1$ . If such a node is found then the RECONSTRUCT mechanism is called again with that particular node.
- lines 28-33, here the node bucket contents are checked and utilized until there is no more element in the node-bucket;

## Chapter 3

### Query Processing for Directed Trees

In this chapter we present algorithms for two types of queries viz. Reachability queries and finding Least Common ancestor for any two given nodes. For other queries we apply the algorithms used for DAGs considering them as special cases of DAG queries. The labeling scheme proposed by H.V. Jagadish [1] has been adapted.

Example:

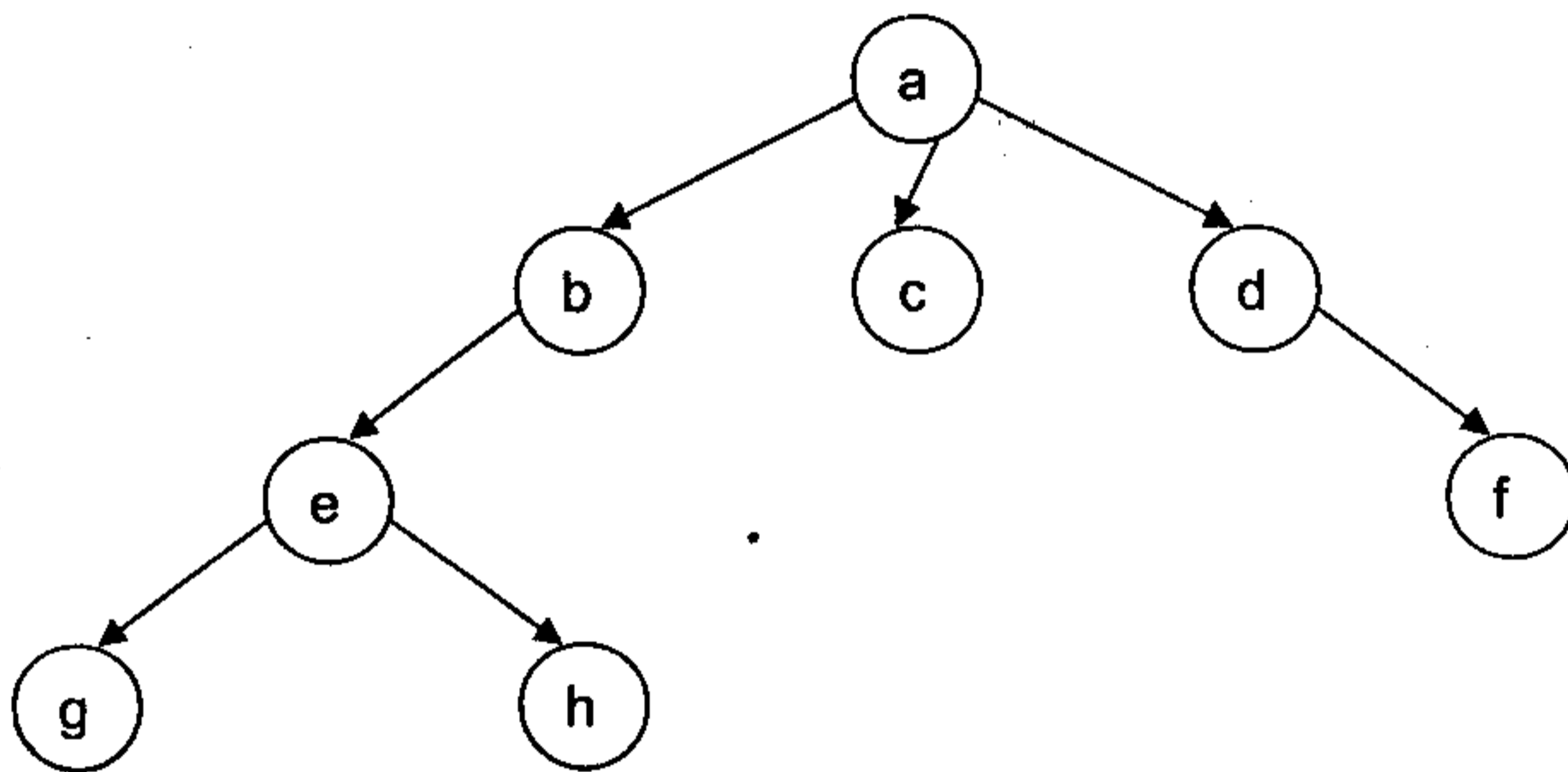


Fig 3.1 Example Directed Tree

#### 3.1 Pre-processing

- Number each node to reflect its relative position in a post order traversal of the tree. Let us call this number a *post-order number*.

Mapping Table for the DAG shown in Fig 3.1

Actual Node Value	a	b	c	d	e	f	g	h
Post-Order Number	8	4	5	7	3	6	1	2

- Now, to each node in the tree assign an index consisting of the lowest post-order number among its descendants. For simplicity, we assume that every node can reach itself, so that the index associated with a leaf node is the same as the post-order number of the node.
- From now onwards, all the processing is done with reference to this post order numbering of the nodes.

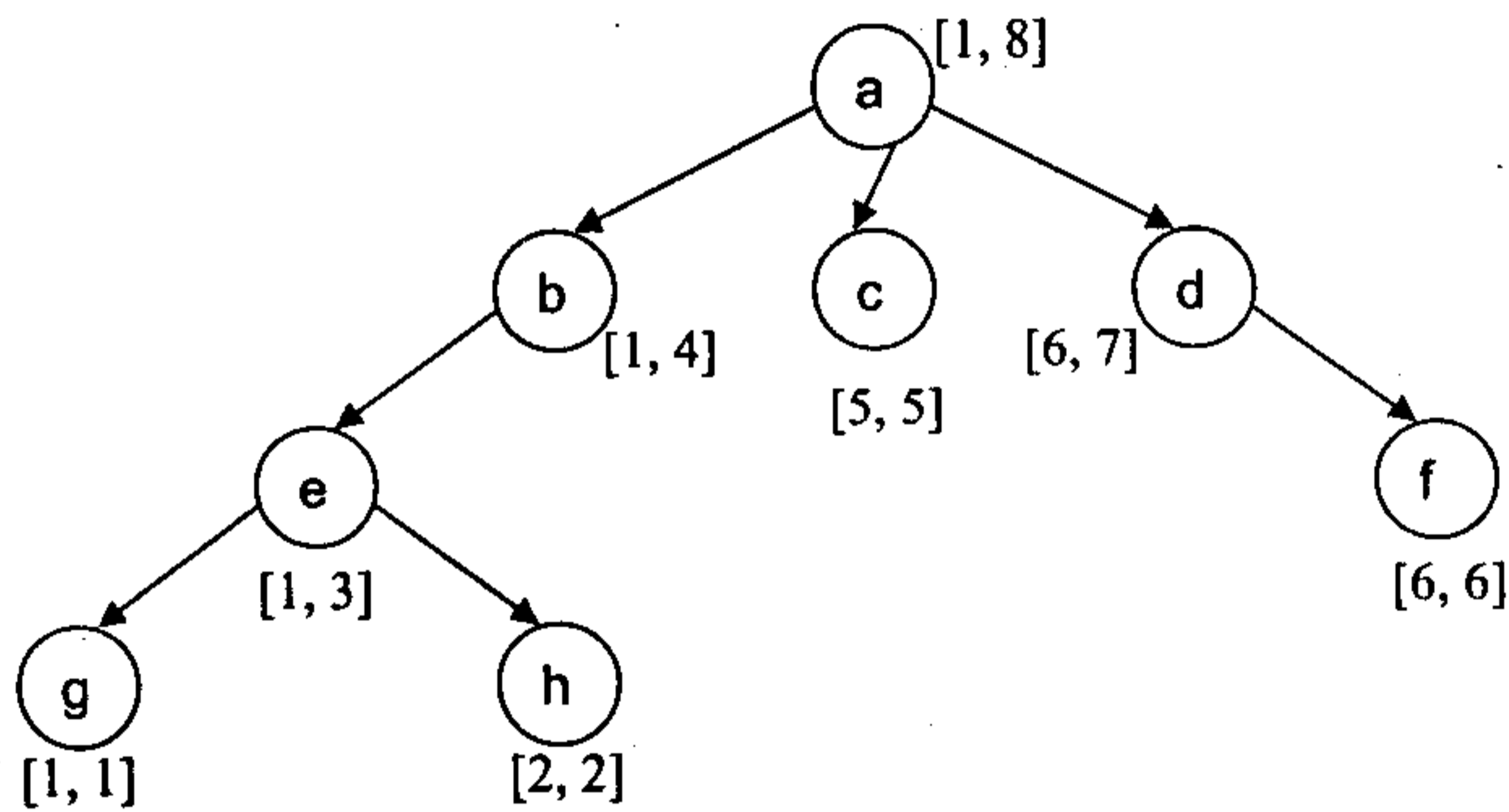


Fig 3.2 Compressed transitive closure for the tree shown in Fig 3.1

- Construct Normalized Paths and Node Buckets.

Normalized paths and node-buckets for the directed tree in Fig 3.1:

1. Normalized Paths

P1: {8, 4, 3, 1}

P2: {3, 2}

P3: {8, 5}

P4: {8, 7, 6}



**2. Node Buckets [Actual Node-Value, Post-order Number, Interval obtained, Paths]**

a: 8 [1, 8] P1(8) P3(8) P4(8)

b: 4 [1, 4] P1(8)

c: 5 [5, 5] P3(8)

d: 7 [6, 7] P4(8)

e: 3 [1, 3] P1(8) P2(3)

f: 6 [6, 6] P4(8)

g: 1 [1, 1] P1(8)

h: 2 [2, 2] P2(3)

- Note that in the first line 8 represents the post order number of 'a', 1, 8 in the square brackets corresponds to its in-degree and out-degree respectively and the number 8 in the parenthesis represents the post order number of the start node of that path.
- It is to be noted that, while traversing the DFS we must choose the branches as they are done in post-order traversal of the tree.

### **3.2 Algorithms for query processing**

Note that as mentioned earlier all the arguments passed and return values in the following algorithms correspond to the post order numbers, also assume that the mapping to the post order numbers and node identifiers are done implicitly.

**3.2.1 Reachability:** The intervals that are obtained from the pre-processing step-1 give the reachability information.

```
isReachable (u, v)
{
    if v lies within the interval associated with node u, then return true,
    otherwise return false.
}
```

E.g.:

Query: isReachable (a, g)

Answer: True

**3.2.2 Least Common Ancestor of node U and node V:**

**Definition:** The Least Common Ancestor of nodes U and V in a directed tree is the ancestor of U and V that is located farthest from the root.

```
LCA (u, v)
{
    if (u > v)
    {
        maxPON = u;
        minPON = v;
    }
    else
    {
        maxPON = v;
        minPON = u;
    }
    if(isReachable(maxPON,minPON))
    {
        For each path P for maxPON
```

```

    {
        if (Root(P) != maxPON)
            return the node which comes just before maxPON;
    }
}
else
{
    while (1)
    {
        maxPON = findNext (MaxPON);
        if (isReachable (maxPON,minPON))
            return maxPON;
    }
}
}
findNext (maxPON)
{
    For each path P for maxPON
    {
        if (Root(P) != maxPON)
            return the start node of path P;
    }
}

```

E.g.:

Query: LCA (e, g)

Answer: a

# Chapter 4

## Graph Compression

### *Definitions*

#### **Strongly Connected Component:**

A strongly connected component of a directed graph  $G = (V, E)$  is a maximal set of vertices  $U$  subset or equal to  $V$  such that for every pair of vertices  $u$  and  $v$  in  $U$ ,  $e$  have both  $u \rightsquigarrow v$  and  $v \rightsquigarrow u$ ; that is, vertices  $u$  and  $v$  are reachable from each other.

#### **Hyper-Node:**

All nodes (vertices) of a strongly connected component  $S$  are fused to a single node, called as a hyper-node.

#### **Hyper-Edge:**

If node  $p$ , either a hyper-node or a node outside a strongly connected component  $S$  is connected to more than one node in  $S$  in the same direction  $D$ , all such edges will be fused to only one edge when  $S$  is fused to hyper-node  $H$ . This edge will connect  $p$  and  $H$  in the direction  $D$ . This fused edge will give rise to a hyper-edge.

The network based real life applications considered for our study are represented as directed graphs. One possible way to compress the directed graph to a DAG is by fusing all strongly connected components as hyper-nodes and the information of each hyper-node needs to be maintained.

For finding strongly connected components several algorithms are present in literature [5, 10, 11, 12]. The following linear-time (i.e.,  $\Theta(V+E)$ -time) algorithm computes the strongly connected components of a directed graph  $G = (V, E)$  using two depth-first searches, one on  $G$  and one on  $G^T$  (*Transpose* of graph  $G$ ).

Transpose of a graph  $G = (V, E)$  is defined as  $G^T = (V, E^T)$ , where  $E^T = \{(u, v); (v, u) \in E\}$ . That is,  $E^T$  consists of edges of  $G$  with their directions reversed.

### Algorithm for Fusing directed graph to DAG

Initialize numSCC-Components = 0, SCCNum [i] = 0,  $\forall i \in V$ .

Convert-DirectedGraph-To-DAG (G)

```

{
    1. Initialize a matrix A [numSCC-Components, numSCC-Components] with zeroes.
    2. Call Strongly-Connected-Components (G).
    3. For each edge e (u, v)
        {
            if (SCCNum [u] = SCCNum [v])
            {
                Output the nodes u, v and the edge as components of hyper-node SCCNum [u];
            }
            else
            {
                set A [u, v] = 1;
                (Hyper edges can be obtained easily, by maintaining the list of multiple edges)
            }
        }
    Thus the matrix A becomes an adjacency matrix for the DAG having hyper-nodes and hyper-edges.
    4. Convert the adjacency matrix A to adjacency list;
}

```

Strongly-Connected-Components (G)

```

{
    1. Call DFS (G) to compute finishing times f[u] for each vertex u.
    2. Compute  $G^T$ .
    3. Call DFS ( $G^T$ ), but in the main loop of DFS, consider the vertices in order of decreasing f[u].
    4. Number the vertices of each tree in the depth-first forest of step 3 as a separate
}

```

strongly connected component.

For each tree in the depth-first forest of step 3 do

```
{
    numSCC-Components = numSCC-Components+1;
    for each vertex v of the tree
    {
        SCCNum [v] = numSCC-Components;
    }
}
```

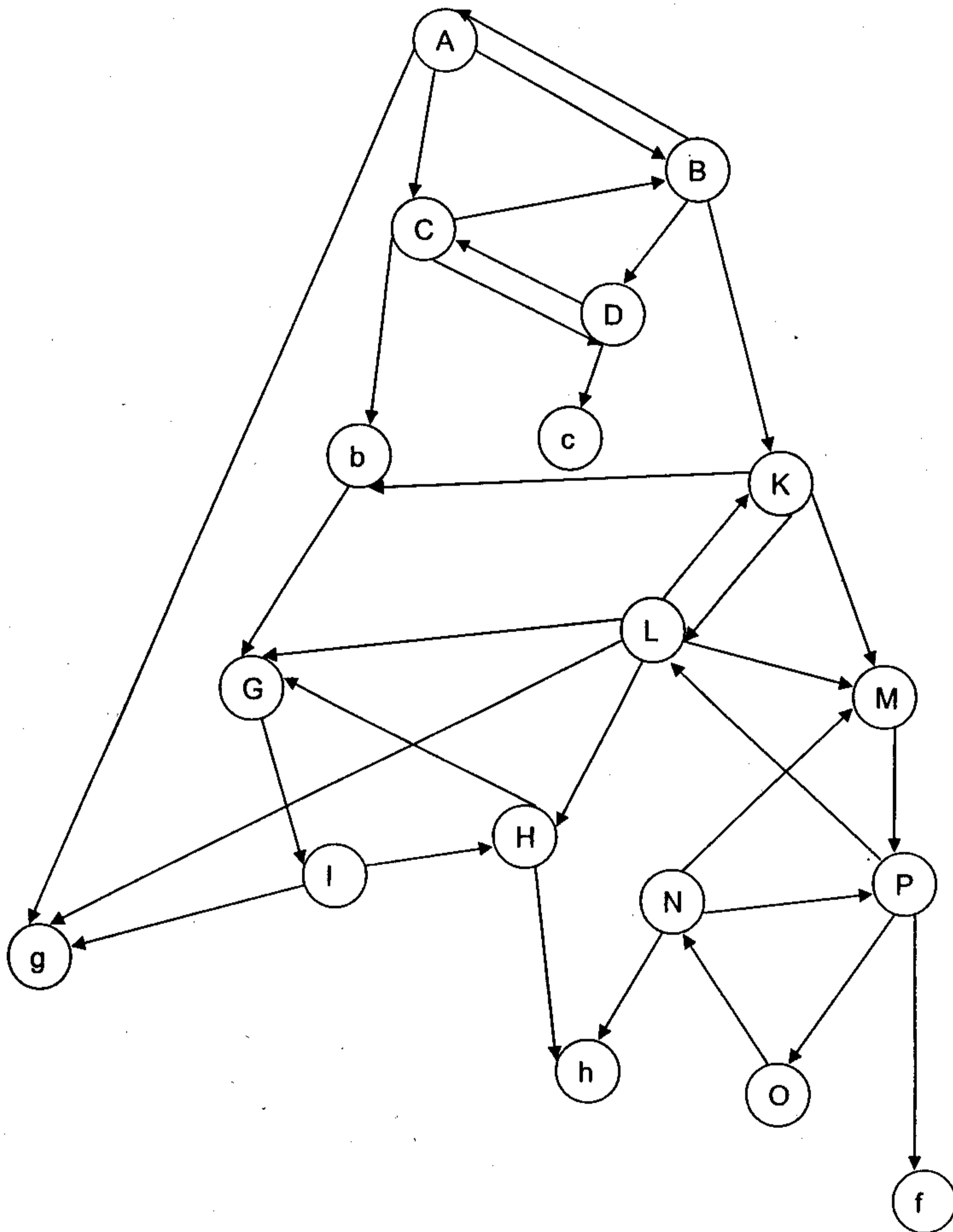
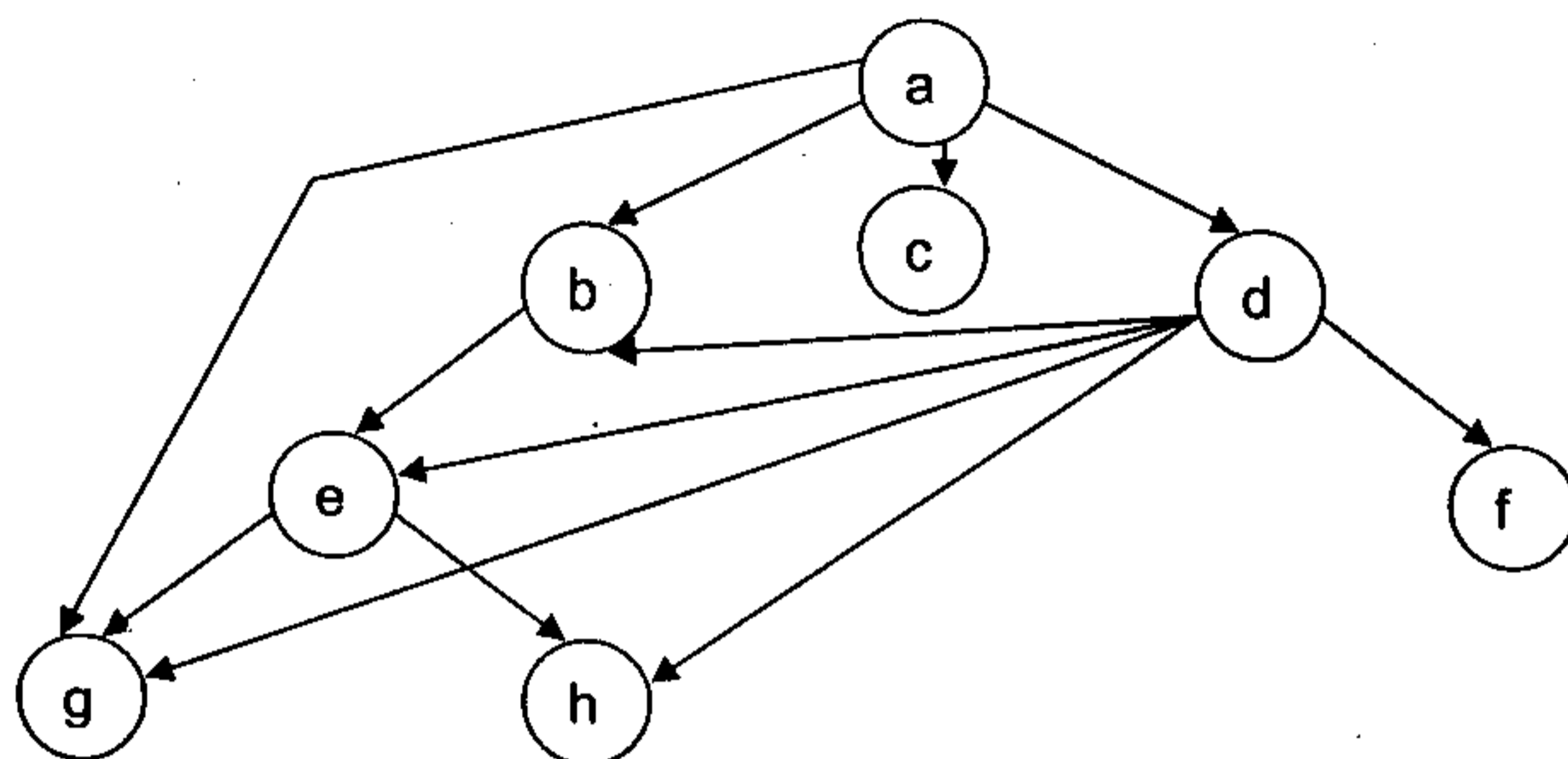


Fig 4.1 Example Directed Graph



**Fig 4.2** DAG that was formed after fusing strongly connected components

After the transformation of the strongly connected components into hyper-nodes we get the DAG shown in Fig 4.2. With reference to the directed graph presented in Fig 4.1, there are 3 hyper-nodes which have been named a, d and e.

Information regarding the hyper-nodes and hyper-edges needs to be maintained to answer the queries, which depend on these hyper-nodes. And the storage and retrieval of these hyper-nodes and hyper-edges is out of the scope of this dissertation work. Information Regarding Hyper-Nodes:

Hyper-Node1, a:

Nodes: {A, B, C, D}

Edges: {(A, B), (A, C), (B, D), (B, A), (C, B), (C, D), (D, C)}

Hyper-Node2, d:

Nodes: {K, L, M, N, O, P}

Edges: {(K, L), (K, M), (L, K), (L, M), (M, P), (P, O), (P, L), (O, N), (N, M), (N, P)}

Hyper-Node3, e:

Nodes: {G, H, I}

Edges: {(G, I), (I, H), (H, G)}

Here onwards hyper-node and hyper-edge also considered as a node and edge, respectively. Storing and Processing of these hyper-nodes and hyper-edges is out of the scope of this dissertation work.



## Chapter 5

### Query Processing for DAGs

Once we got DAG from directed graph by fusing the strongly connected components, pre-processing of this DAG is required for query processing. This is done as follows:

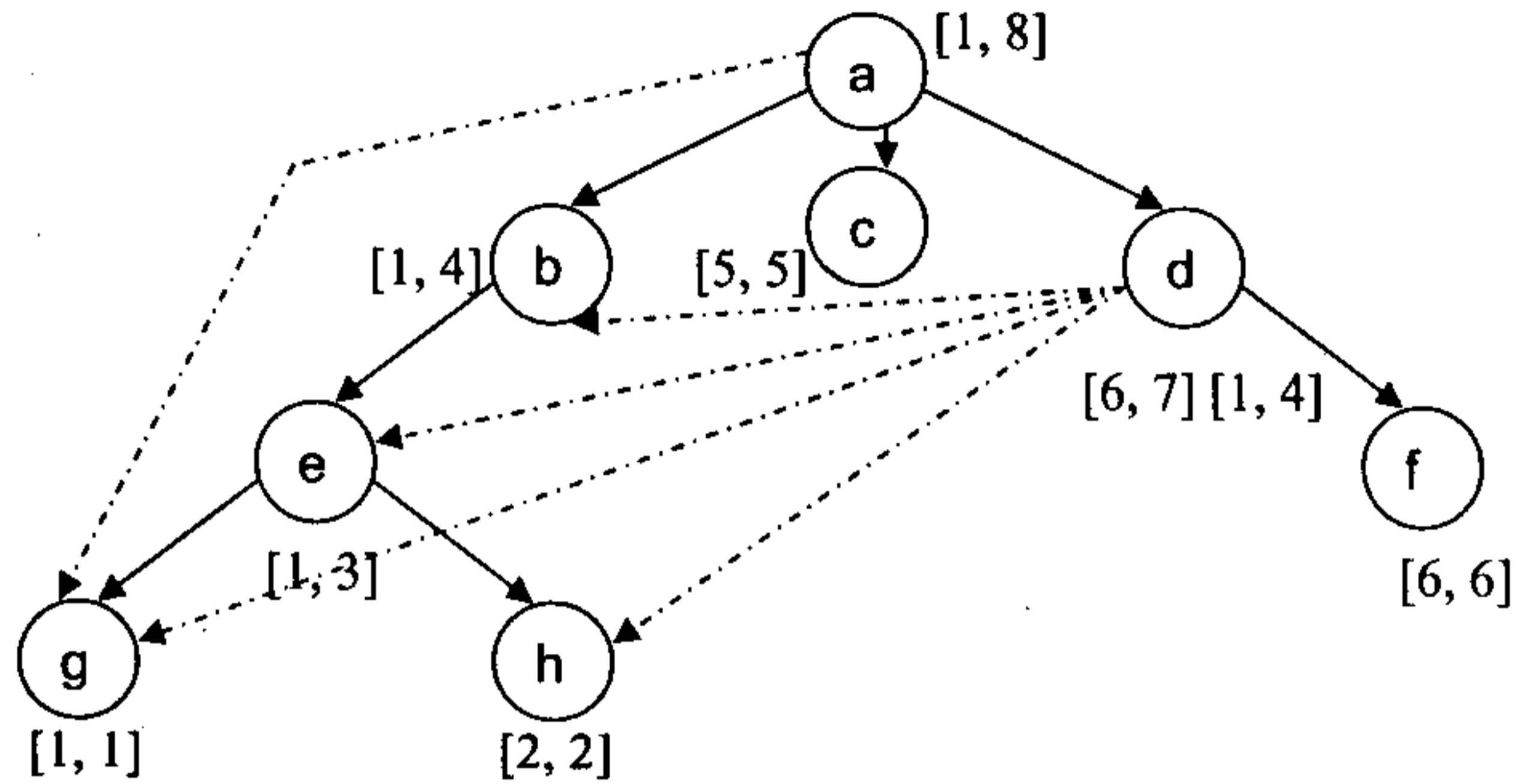
#### 5.1 Pre-processing

By generalizing the numbering scheme that was adopted for directed trees to the case of Directed Acyclic Graphs and assuming that the graph consists of only one connected component; note that disjoint components can be hooked together by creating a virtual root node.

**The compression scheme works as follows:**

1. Find a spanning tree  $T$  for the given graph  $G$ . We call  $T$  the tree-cover of  $G$ .
2. Assign post-order numbers and indices to the nodes of  $T$  as discussed earlier in case of directed trees. Thus, at the end of this step, an interval  $[i, j]$  is associated with each node, such that  $j$  is the post-order number of the node and  $i$  is the lowest post-order number among its descendants.
3. Examine all the nodes of  $G$  in the reverse topological order. At each node  $p$ , do the following processing:
  - For every arc  $(p, q)$ , add all the intervals associated with the node  $q$  to the intervals associated with the node  $p$ .
  - At the time of adding an interval to the interval set associated with a node, if one interval is subsumed by another, discard the subsumed interval. That is, if two intervals  $[i_1, i_2]$  and  $[j_1, j_2]$  are such that  $i_1 \leq j_1$  and  $i_2 \geq j_2$ , then discard  $[j_1, j_2]$ . If two intervals

$[i_1, i_2]$  and  $[j_1, j_2]$  are such that  $j_1 = i_2 + 1$ , then create one  $[i_1, j_2]$  corresponding to these two intervals. Merge two intervals,  $[i_1, i_2]$  and  $[j_1, j_2]$  into  $[i_1, j_2]$ , if  $i_1 \leq j_1 \leq i_2 \leq j_2$ .



**Fig 5.1** Compressed transitive closure for a DAG without finding optimum tree-cover

Mapping Table for the DAG shown in Fig 5.1.

Actual Node Value	a	B	c	d	e	F	g	h
Post Order Number	8	4	7	6	3	5	1	2

- Nodes of a given graph can be covered by more than one spanning tree. However, all tree covers are not equally good. Find optimum tree-cover in the sense that the minimum number intervals associated with nodes (summed over all nodes).

**Algorithm for finding optimum tree-cover:**

1. Topologically sort G.
2. Assume that all nodes with no predecessors are connected to a virtual level 0 root.
3. For every node j in G, in topological order, do:
  - Repeat for each incoming arc pair  $(i_1, j), (i_2, j)$ :
  - if  $\text{size}(\text{pred}(i_1)) > \text{size}(\text{pred}(i_2))$

```

Delete (i2, j)
else
Delete (i1, j)
For every ik immediate predecessor of j
pred (j) = {ik} U pred (ik)

```

pred (j) denotes set of all predecessors of node j, and is computed incrementally.

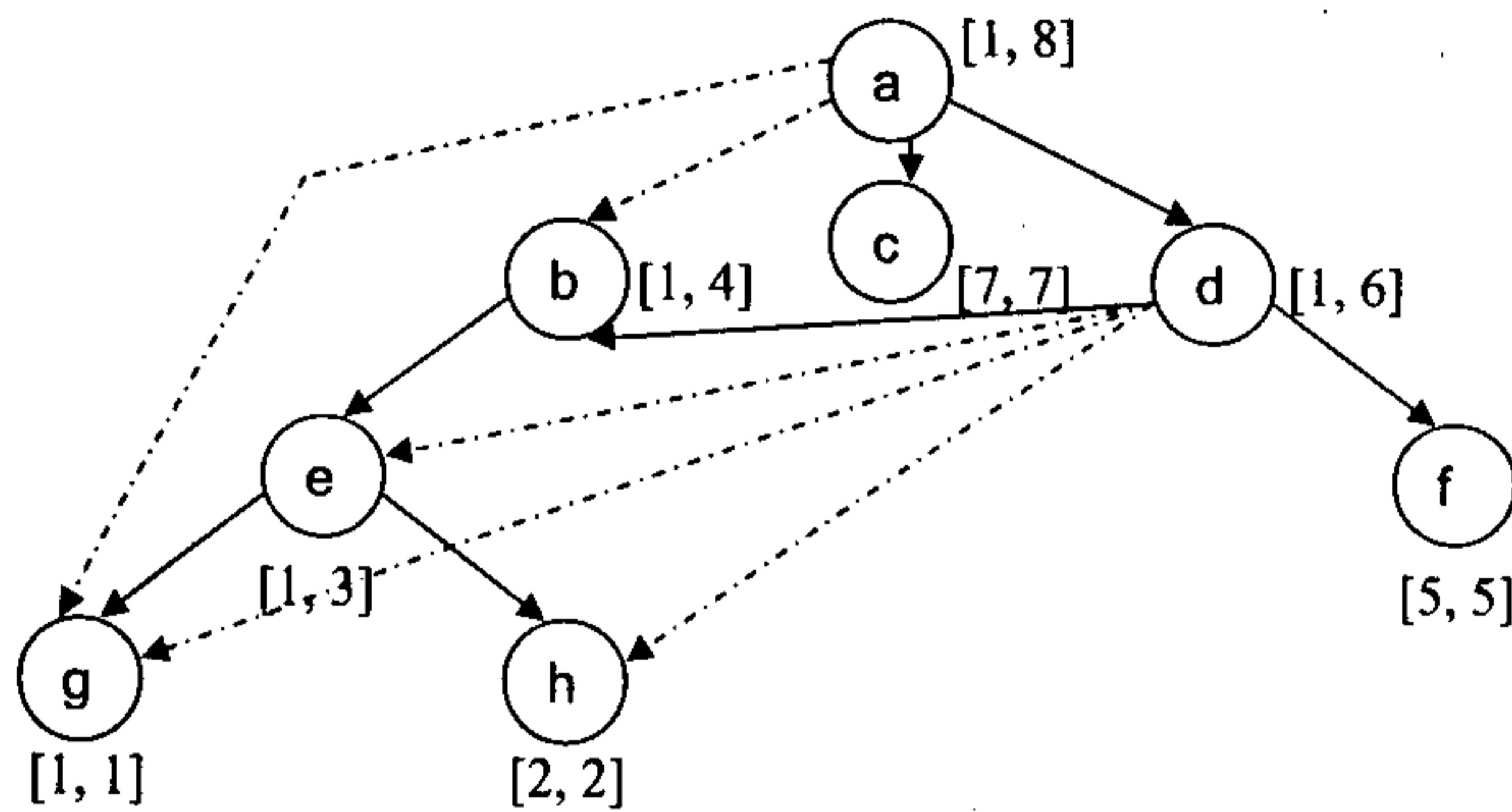


Fig 5.2 Compressed transitive closure for a DAG After finding optimum tree-cover

- Give depth first numbering (DFN) for the DAG.

**Algorithm for finding Depth-first Numbers:**

Initialize dfnValue = 0;

DFNumbering (G)

{

Call DFS (G), in the main loop of DFS, after calculating f[u], calculate

DFN (u) = dfnValue = dfnValue + 1;

}

Modified Mapping Table for the DAG shown in 5.2

Actual Node Value	a	B	c	d	e	F	g	H
Post Order Number	8	4	7	6	3	5	1	2
Depth First Number	8	4	5	7	3	6	1	2

**Need for DFN:**

A numbering scheme is needed to maintain the ancestor-descendent relation. This order is not maintained by the post-order number that we are constructing. This can be seen from the following example:

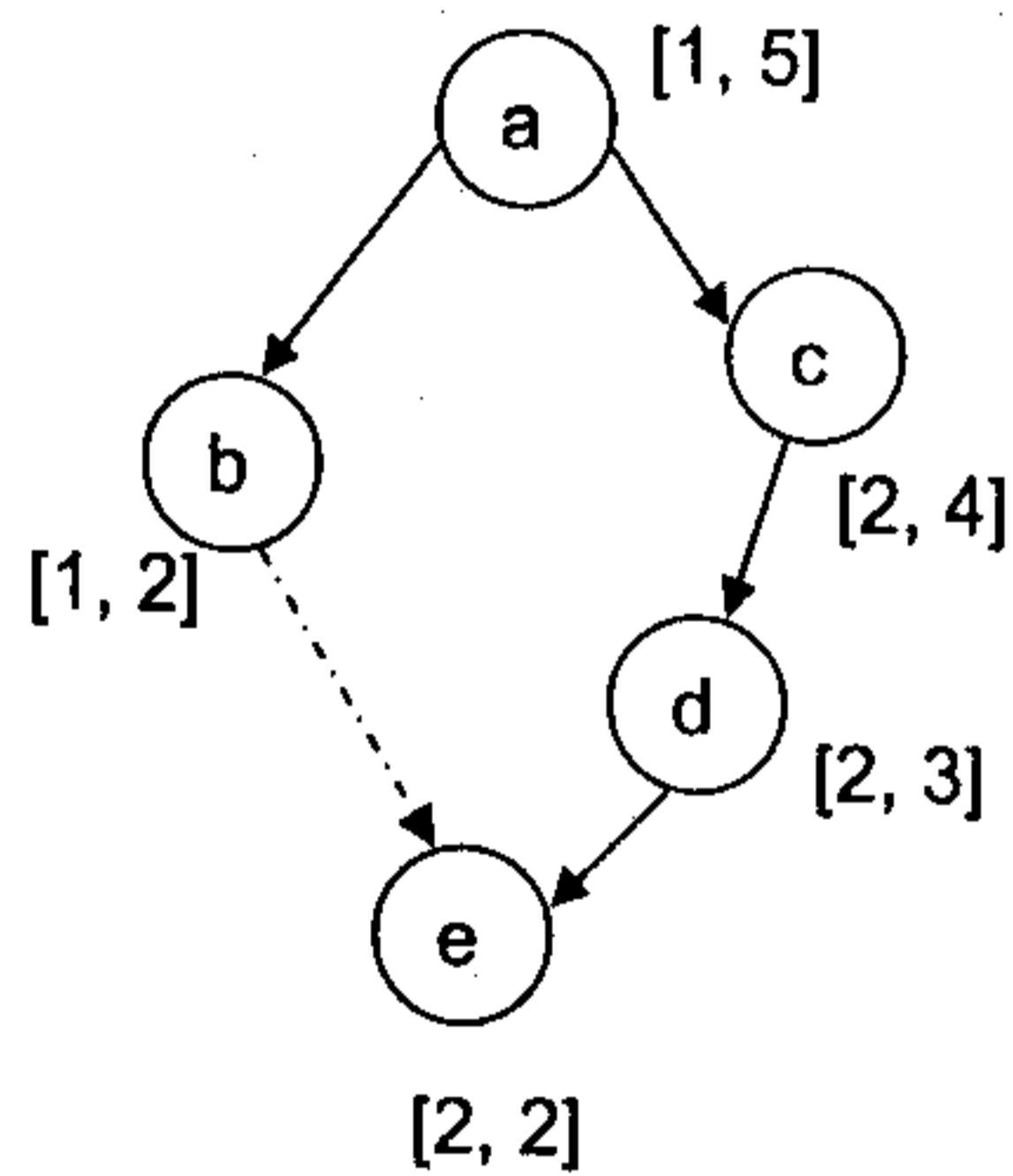
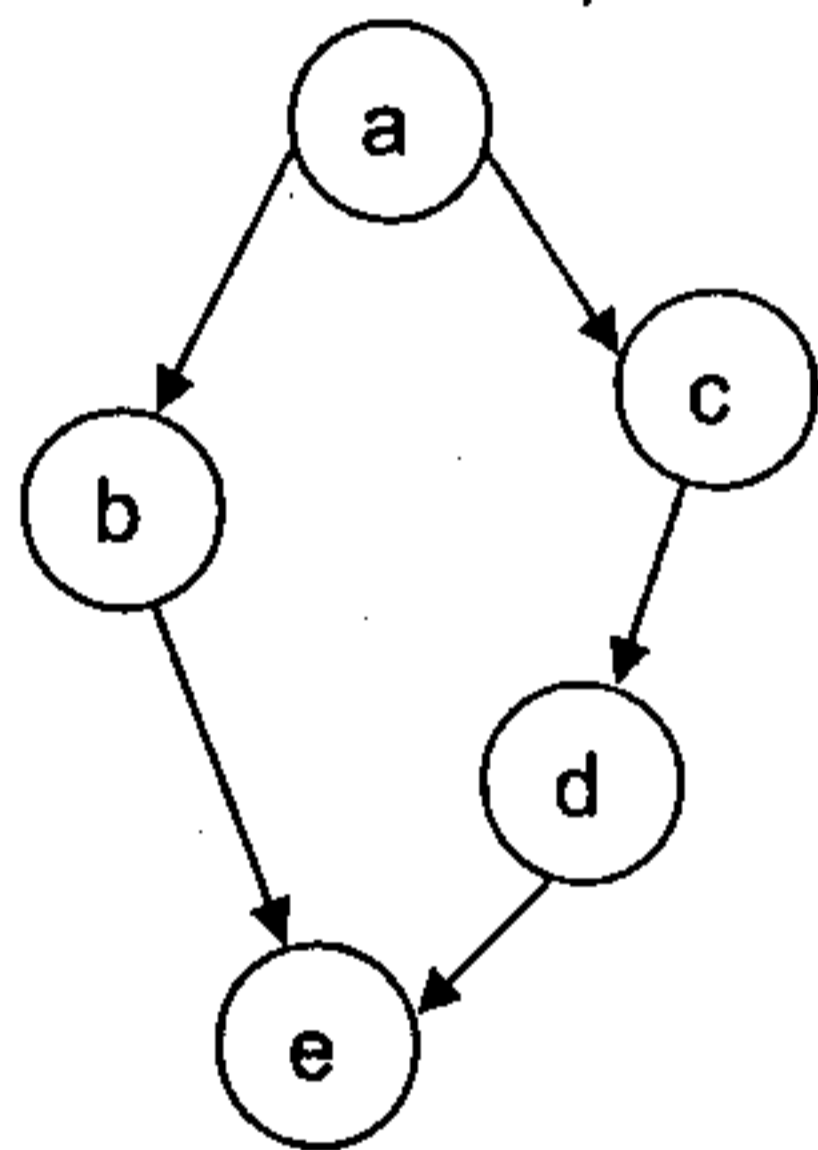


Fig 5.3 Example DAG

Fig 5.4 After Numbering the DAG

Mapping Table for DAG shown in Fig 5.4

Actual Node Value	A	b	c	D	e
Post Order Number	5	1	4	3	2
Depth First Number	5	2	4	3	1

It is to be noted that, while traversing the DFS if we choose the branches as they are done in post-order traversal of the tree then depth-first-number = post-order number.

- Run the modified DFS algorithm on the DAG shown in Fig 2, to construct Normalized Paths and Node Buckets.

Let us take a reasonably complicated example and carry out the steps discussed above. Compressed transitive closure for the DAG shown in Fig 2.1 by adding dummy root node is shown in Fig 5.5.

Mapping Table for DAG shown in Fig 5.5

Node	D	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
PostOrder	18	15	14	13	4	3	2	1	7	6	5	12	10	9	8	11	17	16
DFN	18	15	14	13	6	5	2	1	4	3	7	12	10	9	8	11	17	16

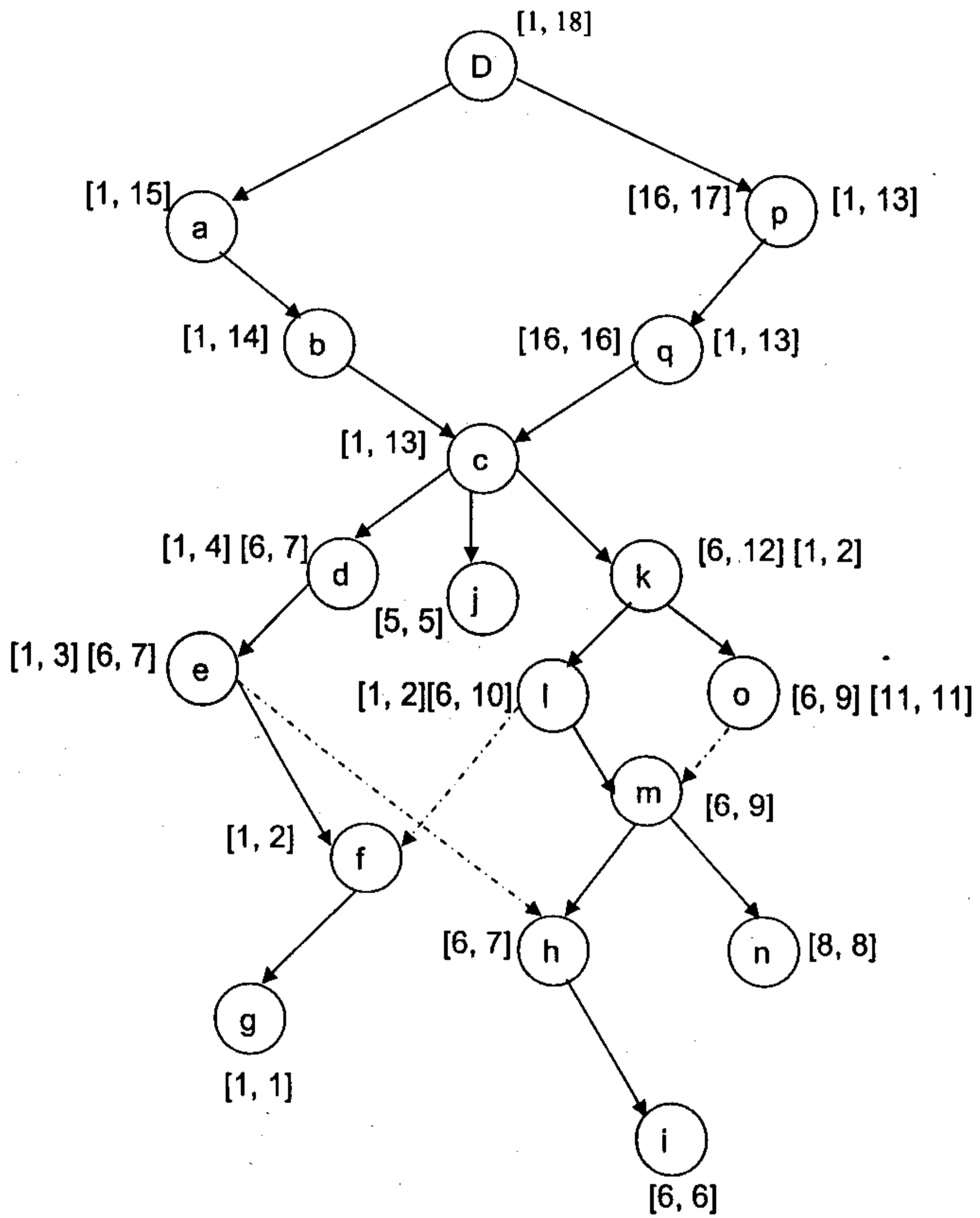


Fig 5.5 Compressed transitive closure for the DAG shown in Fig 2.1 by adding dummy root node

**Normalized Paths:**

P1: [1, 1] {18, 15, 14, 13, 4, 3, 2, 4}

P2: [0, 1] {3, 7, 6}

P3: [0, 1] {13, 5}

P4: [0, 0] {13, 12, 10, 2}

P5: [0, 0] {10, 9, 7}

P6: [0, 1] {9, 8}

P7: [0, 0] {12, 11, 9}

P8: [1, 0] {18, 17, 1, 13}

**Node Buckets:**

D:	18	18	[0, 2]	[1, 18]	P1 (18, 1), P8 (18, 13)
a:	15	15	[1, 1]	[1, 15]	P1 (18, 1)
b:	14	14	[1, 1]	[1, 14]	P1 (18, 1)
c:	13	13	[2, 3]	[1, 13]	P1 (18, 1), P3 (13, 5), P4 (13, 2), P8(18, 13)
d:	6	4	[1, 1]	[1, 4] [6, 7]	P1 (18, 1)
e:	16	16	[1, 1]	[16, 16] [1, 13]	P8 (18, 13)
f:	2	2	[2, 1]	[1, 2]	P1 (18, 1), P4 (13, 2)
g:	1	1	[1, 0]	[1, 1]	P1 (18, 1)
h:	4	7	[2, 1]	[6, 7]	P2 (3, 6), P5 (10, 7)
i:	3	6	[1, 0]	[6, 6]	P2 (3, 6)
j:	7	5	[1, 0]	[5, 5]	P3 (13, 5)
k:	12	12	[1, 2]	[6, 12] [1, 2]	P4 (13, 2), P7 (12, 9)
l:	10	10	[1, 2]	[1, 2] [6, 10]	P4 (13, 2), P5 (10, 7)
m:	9	9	[2, 2]	[6, 9]	P5 (10, 7), P6 (9, 8), P7 (12, 9)
n:	8	8	[1, 0]	[8, 8]	P6 (9, 8)
o:	11	11	[1, 1]	[11, 11] [6, 9]	P7 (12, 9)
p:	17	17	[1, 1]	[16, 17] [1, 13]	P8 (18, 13)
q:	16	16	[1, 1]	[16, 16] [1, 13]	P8 (18, 13)

- consider that 5<sup>th</sup> row, here for the node 'd', 6 denotes DFN, 4 denotes the post order number, [1,1] in column 4 represents in-degree and out-degree of d, [1, 4] [6, 7] represents the tree and non-tree intervals, in the last column Pi represents the path identity and information in the parenthesis represents the start node and end node of the corresponding path.

## 5.2 Query Processing

### 5.2.1 Reachability:

The intervals that are obtained from the pre-processing step-1 give the reachability information.

```
isReachable (u, v)
{
    if v lies in any one of the interval corresponding to node u then return true
    else return false;
}
```

E.g.:

Query: **isReachable (k, g)**

Answer: **True**

### 5.2.2 Find all ancestors of a given node X:

Initialize Array GlobalColorFlag with WHITE;

Ancestors (node X, Array AncestorListFlag)

```
{
    if (GlobalColorFlag (X) = BLACK)
    {
        Return;
    }
    For each path P in the node Bucket of X do
    {
        if (root (P) = X)
            Continue;
        else
        {
            Push all elements in path P up to node X;
            (Push an element only if it is not pushed already)
            While (stackTop = -1)
```



```

        {
            Y = pop ();
            AncestorListFlag (Y) = 1;
            Ancestors (Y, AncestorListFlag);
        }
    }
    GlobalColorFlag (X) = BLACK;
}

```

E.g.:

Query: **Ancestors (f)**

Answer: **a, p, b, q, c, d, j, k, e, l**

### 5.2.3 Find all common ancestors of nodes X and Y:

CommonAncestors (node X, node Y)

```

{
    Initialize Array GlobalColorFlag with WHITE;
    Ancestors (X, AncestorListFlag1);
    Initialize Array GlobalColorFlag with WHITE;
    Ancestors (X, AncestorListFlag2);
    for each vertex  $i \in V(G)$ 
        CommonAncestorsFlag (i) = AncestorListFlag1 (i) + AncestorListFlag2 (i);
    Output all nodes whose CommonAncestorsFlag (i) = 2;
}

```

E.g.:

Query: **CommonAncestors (d, j)**

Answer: **a, p, b, q, c**

### 5.2.4 Find all descendants of a given node X:

Descendants (node X, Array DescendantsListFlag)

```

{
    For each node Z in the interval list except X;

```

```

    {
        DescendantsListFlag (Z) = 1;
        Output Z as Descendant of X;
    }
}

```

E.g.:

Query: **Descendants (i)**

Answer: **f, m, g, h, n, i**

### 5.2.5 Find all common descendants of nodes X and Y:

AllCommonDescendants (node X, node Y)

```

{
    Descendants (X, DescendantsListFlag1);
    Descendants (Y, DescendantsListFlag2);
    for each vertex  $i \in V(G)$ 
        CommonDescendantsFlag (i) = DescendantsListFlag1 (i) +
            DescendantsListFlag2 (i);
    Output all nodes whose CommonDescendantsFlag = 2;
}

```

E.g.:

Query: **CommonDescendants (l, o)**

Answer: **m, h, n, i**

### 5.2.6 Find Least Common Ancestors for any given pair of nodes X and Y:

**Definition:** Let  $G = (V, E)$  be a DAG, and let  $x, y \in V$ . Let  $G_{x,y}$  be the sub-graph of  $G$  induced by the set of all common ancestors of  $x$  and  $y$ . Define  $SLCA(x, y)$  to be the set of out-degree 0 nodes in  $G_{x,y}$ . Then the least common ancestors of  $x$  and  $y$  are exactly the elements of  $SLCA(x, y)$ .

FindAllAncestorsModified (node Y, Array AncestorListFlag)

```

{
    LocalFlag = 0;

```

```

if (GlobalColorFlag (Y) = BLACK)
{
    if (AncestorListFlag (Y) = 3)
        AncestorListFlag (Y) = 2;
    Return;
}
For each path P in the node Bucket of Y do
{
    if (root (P) = Y)
        Continue;
    else
    {
        LocalFlag = 0;
        Push all elements in path P up to node Y;
        (Push an element only if it is not pushed already)
        While (stackTop = -1)
        {
            Z = pop ();
            if (AncestorListFlag (Z) = 1 )
            {
                if (GlobalFlag = 0)
                {
                    GlobalFlag = 1;
                    GlobalNode = Z;
                    if (LocalFlag = 0)
                    {
                        LocalFlag = 1;
                        //3 CA and Probable NCA
                        AncestorListFlag (Z) = 3;
                    }
                    else
                        AncestorListFlag (Z) = 2;
                }
            }
        }
    }
}

```

```

        FindAllAncestorsModified (Z, AncestorListFlag);
    }
}
GlobalColor (Y) = BLACK;
if (GlobalNode = Y)
    GlobalFlag = 0;
if (GlobalFlag = 1)
{
    if (AncestorListFlag (Y) != 0)
        AncestorListFlag (Y) = 2; // Surely not a NCA but CA
}
}
LeastCommonAncestors (node X, node Y)
{
    Initialize GlobalColorFlag with WHITE;
    Ancestors (X, AncestorListFlag);
    Initialize GlobalColorFlag with WHITE;
    FindAllAncestorsModified (Y, AncestorListFlag);
    for each vertex  $i \in V(G)$ 
    Output all nodes whose AncestorListFlag (i) = 3 as NCA and all nodes
    whose AncestorListFlag (i) > 1 as CommonAncestors for nodes X and Y;
}

```

E.g.:

Query: LeastCommonAncestors (h, f)

Answer: e, l

### 5.2.7 Find all the paths between the nodes U and V:

1. if (V is not reachable from u)

{

    No Path, Exit;

}

2. else

{

Step2.1 For Node U

```
{  
  Retrieve all the paths for U from the Node Bucket.  
  For each path P  
  {  
    if (Root(P) = U)  
    {  
      if (V is Reachable from second(P))  
      {  
        Path List [P] = 1;  
        Set the Natural Root field True;  
      }  
      else  
        Path List [P] = 0;  
    }  
    else if (Leaf(P) = U)  
    {  
      Path List [P] = 0;  
    }  
    At most One Path P1 will be left for U  
    If such path P1 exists  
    {  
      if (V is reachable from next (U))  
      {  
        Remove all nodes before U in path P1.  
        Set Root Field to U;  
        Path List [P1] = 1;  
      }  
    }  
  }  
}
```

Step2.2 Find all the Anchor Nodes X between U and V using DFN such that X is reachable from U and V is Reachable from X;

```
For each X  
{
```

```

if (In Degree[X] = 1 && Out Degree[X] >1)
    Type [X] = 1;
else if (In Degree[X] > 1 && Out Degree[X] =1)
    Type [X] = 2;
else if(In Degree[X] > 1 && Out Degree[X] >1)
    Type [X] = 3;
}
For each Anchor Node X in decreasing DFN
{
    Retrieve all the Paths P for X
    If (Type [X] = 3)
    {
        For Each path P
        if (Path List[P] = -1)
        {
            if (Leaf(P) = X)
                Path List [p] = 0;
            else if(Root(P) = X)
            {
                if (V is Reachable from second(P))
                    Path List [P] = 1;
                else
                    Path List [P] = 0;
            }
        }
        else if(Path List[P] = 1 and Leaf(P) != X)
        {
            if (V is not Reachable from next(X))
                Remove all the elements after X;
        }
    }
}
For all the remaining Paths (at most one path will remain)
{
    if (V is reachable from next (X))
    {
        Remove all nodes before U in path P1.
    }
}

```

```

        Append this path to any path P2
        whose (Leaf = X and Path List[P2] = 1);
    }
    Path List [P] = 1;
}
}
else if(Type[X ] = 2)
{
    For Each path P
    if (Path List[P] = -1 and Leaf(P) = X)
        Path List [p] = 0;
    for all the remaining Paths (at most one path will remain)
    {
        if (V is reachable from next (X))
        {
            Remove all nodes before U in path P1.
            Append this path to any path P2
            whose (Leaf = X and Path List [P2] = 1);
            Path List [P] = 1;
        }
    }
}
else if(Type[X ] = 1)
{
    For Each path P
    if (Path List[P] = -1 and Root(P) = X)
    {
        if (V is Reachable from second(P))
            Path List [P] = 1;
        else
            Path List [P] = 0;
    }
    else if (Path List[P] = 1 and Leaf(P) != X)
    {
        if (V is not Reachable from next(X))

```

```

                                Remove all the elements after X;
                                }
                                }
                                }
Step2.3 for Node V
    {
        for each path P for V
        if (Path List[P] = 1 and Leaf(P) != V)
        {
            Remove all nodes after V in that path P.
            //at most one such path exist.
        }
    }
}

```

According to the modifications that are done above Natural Root, Natural Leaf, in-degree, out-degree and path origin and path destination fields has to be modified and giving the resulting paths to the reconstruction algorithm.

E.g.:

Query: **AllPathsBetweenUandV (b, f)**

Answer: **b-c-d-e-f**

**b-c-k-l-f**



## **Chapter 6**

### **Conclusion and Future Work**

This dissertation thus constitutes graph compression i.e., conversion from directed graphs to DAGs, path normalization process, construction of node buckets for normalized paths and algorithms for processing the path queries on trees and DAGs. The postulates and methods adopted have not been proved but intuitively accepted.

As a further study, we suggest the following:

1. Rigorous study of storage and indexing of node-buckets is necessary.
2. Storage and processing of hyper-nodes is yet to be explored.
3. The present work considers only static directed acyclic graphs, so the study of dynamic graphs remains to be attended.

## Bibliography

- [1] R. Agrawal , A. Borgida , H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases", Proceedings of the 1989 ACM SIGMOD international conference on Management of data, p.253-262, June 1989, Portland, Oregon, United States 262, June 1989, Portland, Oregon, United States.
- [2] A Bagchi, S. Mitra and A. K. Bandyopadhyay, "SONSYS-A system to model a social network," In Proceedings 6<sup>th</sup>.International Conference on Information Technology, pp.371-374, Bhubaneswar, India, Dec.2003.
- [3] Christophides, Vassilis and Plexousakis, Dimitris and Scholl, Michel and Tourtounis, Sotirios, "On Labeling Schemes for the Semantic Web", In Proceedings International WWW Conference (2003), Budapest, Hungary.
- [4] Michael A. Bender, Giridhar Pemmasani, Steven Skiena, Pavel Sumazin, Finding Least Common Ancestors in directed acyclic graphs, proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, P.845-854, January 07-09, 2001, Washington, D.C., United States.
- [5] R. Tarjan. "Depth first search and Linear Graph Algorithms", SIAM Journal of Computing, 1(2):146-160, June 1972.
- [6] B.Amann and M.Scholl, "Gram: a graph data model and query languages", *Proc.ACM Conf. On Hypertext*, pp. 201-211, 1992.
- [7] D. Shasha, J. T. L. Wang, and R. Giugno. "Algorithmics and applications of tree and graph searching", In Symposium on Principles of Database Systems, pages 39--52, 2002.
- [8] L. Chen, A. Gupta and M. E. Kurul, "Efficient pattern matching on Directed Acyclic Graphs", Proc.IEEE ICDE, 2005, also in <http://gupta@sdsc.edu>
- [9] N. Bruno, N. Koudas and D. Srivastava, "Holistic twig joins: Optimal XML pattern matching", Proc.SIGMOD, pp.310-321, 2002.

[10] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction to algorithms", *MIT Press*, 1990.

[11] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structure and Algorithms", Addison-Wesley, Reading, Mass. 1983.

[12] Esko Nuutila and Eljas Soisalon-Soininen, "On Finding the Strongly Connected Components in a Directed Graph", *Information Processing Letters*, 49(1), 1994.

[13] <http://www.geneontology.org>

[14] <http://www.uspto.gov/go/classification/>