

M. Tech (Computer Science) Dissertation Series

APPROXIMATE SCHEMA DESIGN BY CONCEPTUAL CLUSTERING

A dissertation submitted in partial fulfilment of the requirements for the **M.Tech. (Computer Science)** degree of Indian Statistical Institute

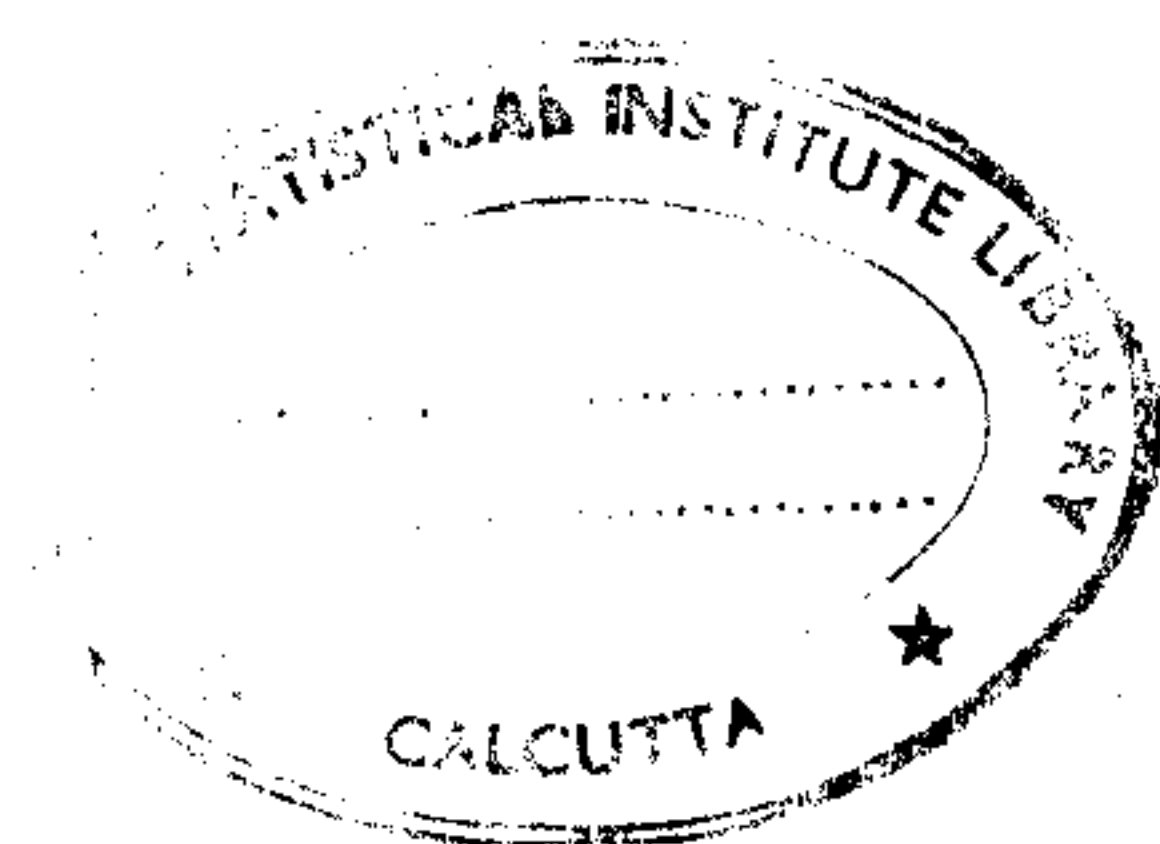
By

Subhamoy Maitra

under the supervision of

Aditya Bagchi
Computer and Statistical Services Centre
INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road
Calcutta-700035

1996




Certificate of Approval

This is to certify that the thesis entitled *Approximate schema design by conceptual clustering* submitted by *Subhamoy Maitra*, in partial fulfilment of the requirements for *M. Tech. in Computer Science* degree of the *Indian Statistical Institute, Calcutta*, is an acceptable work for the award of the degree.

Date: July 26, 1996.


(Supervisor)


(External Examiner)

Acknowledgement

My sincerest gratitude goes to *Prof. Aditya Bagchi* for his guidance, advice, enthusiasm and support throughout the course of this dissertation.

I would also like to take this opportunity to thank *Prof. Bhargab B. Bhattacharya*, *Prof. K. Sikder*, *Prof. B. P. Sinha*, *Dr. Bimal Roy* and *Dr. Subhash Nandi* for the excellent courses they have offered in M.Tech. which helped me in this work.

Thanks to *Dr. Probal Sengupta*, *Mr. Subhash Kundu* and *Dr. Partha Pramanik* who encouraged the research by giving proper technical advice whenever required.

Mr. Pranab Chakraborty, *Mr. Kausik Dutta*, *Mr. Pinakpani Pal*, *Mr. Sounak Misra* and *Mr. Paramartha Dutta* played very crucial role in contributing numerous suggestions throughout the project. I thank them all and also my other classmates who had made my two years stay at ISI enjoyable.

I express my heartfelt thanks to the members of the M.Tech Dissertation Committee.

Finally, I express my gratitude to my parents and brother for whom I did not have to face any trouble during this period.

Contents

1	Introduction	3
1.1	Salient features of the proposed system	4
1.1.1	Initial Assumption	4
1.1.2	Expected Output	4
1.1.3	Additional Features	5
2	Present Scenario	6
2.1	Classification and knowledge representation	6
2.1.1	Classification	6
2.1.2	Semantic Nets	6
2.2	Decision Trees	6
2.3	Database Schema Design	7
2.4	Vertex Connectivity - A Concept from Graph Theory	8
2.5	Remarks	8
3	Learning and verification	9
3.1	Learning	9
3.1.1	Data Structure	9
3.1.2	Algorithm	9
3.1.3	Correctness of the algorithm	11
3.1.4	Example	11
3.2	Verification	12
3.2.1	Completeness	12
3.2.2	Uniqueness	12
3.2.3	Checking with examples	13
3.3	Remarks	13
4	Classification	14
4.1	Different matching concepts	14
4.1.1	Path identification and exact match	14
4.1.2	Perfect match	15
4.2	Break	16
4.2.1	Restricted vertex connectivity	16
4.2.2	Problem Description	17
4.2.3	Mapping	18

4.3	Fuse	19
4.3.1	Role of Domain Expert	19
4.4	Add	19
4.4.1	Partially Classified Leaf	20
4.5	Tree Restructuring	20
4.6	Example	21
4.7	Remarks	22
5	Classification with some of the attribute values unknown	23
5.1	Forced match	23
5.2	Best match	24
5.3	Tree Restructuring	25
5.3.1	Breaking of leaf level class	26
5.4	Remarks	26
6	Application and Future possibilities	27
6.1	Database Applications	27
6.2	Future Work Direction	28
6.3	Conclusion	28

Chapter 1

Introduction

Class and class hierarchy play a very important role in the database schema design. As a modeling tool class provides a generic concept and instances of such concept are the members of the class. Inter-class relationship is represented by the class hierarchy. Usually a data model does not provide the flexibility of inter-node relationships present in a knowledge representation tool like *semantic network*. It only takes care of **IS-A** relationship by inclusion dependencies and **PART-OF** relationship by complex objects. In most of the application areas the intention of the internal description of a class and the class hierarchy for the domain are fixed beforehand. The class of an instance is specified during its insertion to the system and it also strictly conforms to the internal description. However, in a real life situation such rigid class description and hierarchy may not always be able to accept all instances. For example, the class description of *Person* may have *Address* as an attribute. Now, a *Vagabond* does not have an *Address*, nevertheless he is a *Person*. There are several ways to tackle this situation:

1. Mark all the instances of *Vagabond* as exceptions to *Person*.
2. Remove *Address* from the description of *person* so that it does not remain an essential feature for the concept of person any more.
3. Break the class *Person* into two subclasses; one with the *Address* where the existing instances will be placed and the other without the attribute *Address* where new instances of *Vagabond* will be placed.

The standard data model do not provide any of the above features. However, in order to handle a real life application properly, a data model should have the facilities to:

1. Flag exceptions to the class.
2. Change the internal description of the class.
3. Create new classes and reorganize the class hierarchy.

So it is evident that internal class description would contain less and less information as the number of instances increases. A class description should contain only those essential attributes which are common among all the instances. Such *identifying feature set* may again vary with the appearance of exceptions and the class hierarchy may have to be reorganized. In a real life situation, therefore, a

class description will always be approximate and the class hierarchy may have to be dynamic.

In some application areas like Anthropology, Archaeology etc., the classification of instances and class hierarchy are not very well-defined. New instances are often found for which the class descriptions and hierarchy need to be changed. New classes may have to be created. The system may start with an incomplete class hierarchy and initial class descriptions and may keep on modifying them as new instances are inserted. Thus the schema design process would behave like a learning system. One such conceptual clustering algorithm for approximate schema design has already been proposed by Beck et. al.[7]. Their proposal is based on *Explanation Based Learning* and efforts have been made to update class hierarchy after every instance received as *exception*. In an application domain where the number of attributes as well the number of instances are sufficiently large, such an approach may not be very helpful.

This dissertation starts with an incomplete class description. A set of unambiguous learning examples are taken where targeted classes are known. These examples are used to learn the identifying attributes for each class. During actual insertion of new instances the class description, i.e. associated feature set and class hierarchy may be augmented. New classes may be created. An instance may be classified as an *exception* to an existing class. Depending on the application domain and under the advice of domain experts, when the number of *exceptions* exceeds a threshold, a class may be broken into two subclasses or a new class may be added.

An application domain like Archaeology may give rise to instances where incomplete set of attributes may be available. Since an Archaeological material (e.g. a stone statue or a terracotta sample) may be found broken during excavation, insertion of such an instance with some attributes unavailable has also been treated in this dissertation. Different criteria have been defined to identify the best possible match among the existing classes. Such classification would not only be approximate but may be incomplete as well.

1.1 Salient features of the proposed system

1.1.1 Initial Assumption

1. Incomplete class hierarchy available.
2. Complete set of attributes available.
3. Attribute values are True(1), False(0) or Unavailable(X).
4. A set of unambiguous learning samples are available that cover all the classes specified in 1.

1.1.2 Expected Output

1. A class hierarchy that covers (classifies or flags as exception) all examples (instances) received.
2. Each class is associated with updated set of *identifying attributes*.
3. Under classification property indicated.

4. Examples with unavailable data maximally matched.

1.1.3 Additional Features

1. Exploring the possibility of unique classification even with unavailable data.
2. Improvement in classification if some unavailable data are made available.

It should be noted that this system will only work over the identifying attributes. The methods described here will never consider the non-identifying attributes as the features of the classes. So throughout the discussion the terms *attribute* and *identifying attribute* will carry the same meaning. The case is also same for *supervisor* and *domain expert*. We also never allow multiple inheritance in the class hierarchy.

The organization of the document is as follows. The next chapter will talk about some existing results and concepts. Chapter 3 will be on learning and verification techniques. Chapter 4 describes the classification technique when all the attribute values are known. The very next chapter considers classification again, but with some unknown attribute values. The last chapter presents a few database application possibilities based on our algorithms and direction towards future research.

Chapter 2

Present Scenario

This chapter would discuss about some background materials to be used in this dissertation. Basic ideas about classification and knowledge representation have been covered. Conceptual clustering as a technique for database schema design has also been discussed. In this dissertation some concepts of graph theory have been exploited. The necessary background material for that purpose has also been furnished here.

2.1 Classification and knowledge representation

2.1.1 Classification

Classification is the process of assigning, to a particular input, the name of a class to which it belongs[14]. The classes from which the classification procedure can choose may be described in many of ways. Their definition will depend on the use to which they will be put. Classification is an important component of many problem-solving tasks. In its simplest form, it is presented as a straightforward recognition task. Before the classification, the classes should be defined. So it is required to isolate a set of features that are relevant to the task domain. Each class is defined as a structure composed of a subset of those features.

2.1.2 Semantic Nets

In a *semantic net*, information is represented as a set of nodes connected to each other by a set of labeled arcs, which represent relationships among the nodes[14]. The power of semantic net lies in the ability of the programs applied to manipulate it to solve problems. One of the early ways that semantic nets were used was to find relationships among objects by spreading activation out from each of two nodes and see where the activation met. This process is called *intersection search*. Semantic net is also used in conceptual dependency and concept learning.

2.2 Decision Trees

The ability to learn classification is fundamental to intelligent behaviour[8]. Concept-learning algorithms can either be incremental or nonincremental. In the second one, the algorithm infers a concept once, based on the entire set of available training instances. On the other hand for incremental case,

the algorithm revises the current concept definition, if necessary, in response to each newly observed training instances. Formally a decision tree can be defined as either:

1. a leaf node (or answer node) that contains a class name, or
2. a non-leaf node (or decision node) that contains an attribute test with a branch to another decision tree for each possible value of the attribute.

Quinlan's ID3 program induces decision trees of the above form. Each training instance is described as attribute-value pairs. The instance is labeled with the name of the class to which it belongs. It is assumed that an instance belongs to one of the two classes: the positive instances, which are examples of the concept to be learned (the target concept), and the negative instances, which are counter examples of the target concept.

Schilmer and Fisher[1986] has developed an algorithm named ID4 to address the incremental construction of decision trees. But there are many concepts that are not learnable by ID4, even though they are learnable by ID3. Utgoff[8] proposed an algorithm called ID5R, which guarantees to build the same decision tree as ID3 for a given set of instances and also uses incremental learning tasks. The ID5R decision tree is defined to be either of:

1. a leaf node (or answer node) that contains
 - (a) a class name, and
 - (b) the set of instance descriptions at the node belonging to the class
2. a non-leaf node (or decision node) that contains
 - (a) an attribute test, with a branch to another decision tree for each possible value of the attribute, the positive and negative counts for each possible value of the attribute, and
 - (b) the set of non-test attributes at the node, each with positive and negative counts for each possible value of the attribute.

2.3 Database Schema Design

Beck et al[7] presented a conceptual clustering algorithm based on current theories of categorization. The algorithm is used to generate and maintain a database schema containing classes that more accurately represent category structures. The algorithm is based on a trade-off between reasoning about the class description (explanation based learning) and reasoning about instances (case-based reasoning). An instance can be added to a class either because it satisfies an intentional class description or it is similar to other instances in a class. An exception condition is raised when an instance is similar to other instances in a class, yet it violates an intentional description of the class. In such a case, the class may be modified to accommodate the new exception. The input to the algorithm are,

1. An existing database schema containing classes and instances arranged in taxonomy
2. A new class or a new instance

and the output is:

1. A modified version of the database schema which incorporates the new class or new instance.

It is reported that a database management system based on the conceptual clustering algorithm is implemented, though the computational complexity of the algorithm has not been studied formally.

2.4 Vertex Connectivity - A Concept from Graph Theory

This section describes some important graph theoretic results from [5], which will be used for the *break* algorithm in chapter 4. Let $G = (V, E)$ be an undirected graph and let $a, b \in V$ be such that $(a, b) \notin E$. Set $S \subseteq V - \{a, b\}$ is an (a, b) *vertex separator* if every path from a to b passes through a vertex of S . In other words a and b belong to different connected components of $G - S$. The minimum cardinality of any (a, b) *vertex separator* is denoted by $N(a, b)$.

Lemma: Let $G = (V, E)$ be an undirected graph and let $a, b \in V$ be such that $(a, b) \notin E$. Then $N(a, b)$ can be computed in time $O(n^{\frac{1}{2}}e)$.

The *vertex connectivity* c of an undirected graph $G(V, E)$ is the minimal connectivity number of any pair of unconnected vertices. More precisely,

$$c = \begin{cases} n - 1 & \text{if } G \text{ is complete} \\ \min\{N(a, b); (a, b) \notin E\} & \text{otherwise} \end{cases}$$

Theorem: Let $G(V, E)$ be an undirected graph and let c be its vertex connectivity. c can be computed in time $O(cn^{\frac{3}{2}}e) = O(n^{\frac{1}{2}}e^2)$.

2.5 Remarks

In our system the schema will always be approximate and dynamic. For learning we could not use the decision tree methods[8] as that forces each class to contain strictly one attribute. In our idea each class may contain one or more than one identifying attributes which will be similar to database classes, and the initial class hierarchy will be supplied by the domain expert. Again the concept described in [7] changes the schema at the insertion of each instance, if required. We, however, change the class structure only when the number of *exceptions* crosses some threshold value depending on application. Though we do not differ much from the standard concepts of classification, the algorithms used in our approach and the corresponding tools are new.

Chapter 3

Learning and verification

The domain expert provides a lattice structure of classes (i.e. a tree structure with classes as the vertices and is-a relationships as directed edges) and some learning instances (each one is attribute-value pair set with the class it belongs). The purpose is to find a solution for complete categorization of the attributes among the classes, available in the lattice structure. It is to be noted that contradictory learning instances should not be given as input. Verification algorithm should be run after the learning stage which would take care of *completeness*, *uniqueness* and of subtle changes required for better classification.

3.1 Learning

In this section it is described how the identifying attributes are allocated to the classes in the tree structure. We give the algorithm, discuss its correctness and then consider an example for better understanding.

3.1.1 Data Structure

With each class we keep information about its <i> parent class, <ii> leftmost child class and <iii> nearest peer class in the right direction. Multiple inheritance is not allowed. Also with each class there exists a boolean flag *new* or *old* and a set of attributes. Initially the flag is *new* and attribute set is empty for each class.

3.1.2 Algorithm

To describe the algorithm we use the following notations:-

The attribute set of a class C is C_A .

The instance attribute set is I_A .

$TEMP_A$ and X_A are two temporary sets.

The *iflag* is the flag for checking intersection possibility.

Now we describe the algorithm:-

```

    Mark all the classes new;
For each learning instance
Begin
  1. Get all the classes on the path from the root to the class
     mentioned in the given instance and make iflag off;
  2. For each class  $C$  on the path starting from root
    2.1. If  $C$  is marked new
      2.1.1.  $C_A = I_A$ ;
      2.1.2. Mark  $C$  old;
    2.2. Endif
    2.3.  $X_A = I_A \cap C_A$ ;
    2.4.  $TEMP_A = C_A - X_A$ ;
    2.5.  $C_A = X_A$ ;
    2.6.  $I_A = I_A - C_A$ ;
    2.7. If  $TEMP_A$  is not empty
      2.7.1. If  $C$  is not a leaf class
        2.7.1.1 If all the children of  $C$  are old
          2.7.1.1.1.  $C_A = C_A \cup TEMP_A$ ;
        2.7.1.2. else (if all the children classes of  $C$  are not old)
          2.7.1.2.1. For all the old children  $CH$  of  $C$ 
            2.7.1.2.1.1.  $CH_A = CH_A \cup TEMP_A$ ;
            2.7.1.2.1.2. Make iflag on;
          2.7.1.2.2. End For
        2.7.1.3. End If
      2.7.2. else (if  $C$  is a leaf class)
        2.7.2.1.  $I_A = I_A \cup TEMP_A$ 
      2.7.3. End If
    2.8. End If
  3. End For
  4. If  $I_A$  is not empty or iflag is on
  5. Begin
    5.1.  $C_A = C_A \cup I_A$ ;
    5.2.  $P = \text{Parent of } C$ ;
    5.3. While there exists more than one children of  $P$  and
           $X_A (= \text{intersection over all the children of } P)$  is nonempty
    5.4. Begin
      5.4.1.  $P_A = P_A \cup X_A$ ;
      5.4.2. For all the children  $CH$  of  $P$ 
        5.4.2.1.  $CH_A = CH_A - X_A$ ;
      5.4.3.  $P = \text{Parent of } P$ ;
    5.5. End
  6. End
End

```


3.1.3 Correctness of the algorithm

To prove the correctness of the algorithm we have to prove that (i) in each step any of the attributes belonging to the instance and any of the classes will not be lost, (ii) each attribute of an instance targeted towards a class will always exist in one of the classes on the path from the root to the target class and (iii) the algorithm is order independent with respect to the instances applied to the system. To prove the first postulate we find that throughout the step 2 we do not loose any of the attributes. From the steps 2.3 to 2.6, the attributes taken out of C_A are kept in $TEMP_A$, and the attributes taken out of I_A are kept in C_A . In the next stage the attributes of $TEMP_A$ are properly distributed to the *old* children classes if those exists or kept in itself as in step 2.7.1.1.1., otherwise those are added to the I_A for the step 5 where the attributes are added to the leaf class itself. The intersection algorithm in the step 5 also never removes any of the attributes present in the tree structure, rather it sends the common attributes towards the parent.

For the second postulate we find that if some attributes are pushed from a class downwards for its children class, it is pushed only towards the *old* deserving candidates as stated in step 2.7. Also from step 2.3 a class can have those attributes only which are at least once targeted towards each of its children classes. Hence an attribute not targeted towards a class can not exist in any of the classes on the path towards the root class. But again from the first postulate we find that the attribute is not lost out of the class hierarchy. Hence each attribute of an instance targeted towards a class will always exist in one of the classes on the path from the root to the target class.

For the third postulate it should also be noted that step 5 is always checked for common attributes among the sibling classes and they are sent to the parent class. So the above two results reveal we find that the learning algorithm gives the same output irrespective of the order of insertion of the instances. That is, the algorithm places the attributes in relevant classes in an order independent way.

3.1.4 Example

Here we give an example for the learning algorithm. Let us consider the following class hierarchy.

We write the learning examples as target class and corresponding attribute set. The universal attribute set is $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. The examples are $D\{1, 2, 3, 6\}$, $F\{1, 4, 5, 8\}$, $G\{1, 4, 5, 9\}$, $E\{1, 2, 3, 7\}$, $D\{1, 2, 6\}$, $D\{1, 2, 3, 6, 9\}$. Now we trace the algorithm corresponding to insertion of the examples with the figure.

1. The path is ABD . $A = \{1, 2, 3, 6\}$, $B = \phi$, $D = \phi$
2. Path ACF . $A = \{1\}$, $B = \{2, 3, 6\}$ as B is the *old* child of A . $C = \{4, 5, 8\}$, $F = \phi$.
3. Path ACG . A remains same. $C = \{4, 5\}$, $F = \{8\}$ as the old child of C . $G = \{9\}$.
4. Path ABE . $A = \{1\}$, $B = \{2, 3\}$, $D = \{6\}$ being the old child of B . $E = \{7\}$.
5. Path ABD . A remains the same. $B = \{2\}$. Since D, E are both *old* children of B , attribute 3 is again added to the class B . So the structure remains the same.
6. Path ABD . A, B remain the same. After considering the class D once with attribute set $\{6\}$, the

attribute 9 remains in the instance. So for step 5, the attribute will be added to the class *D*.

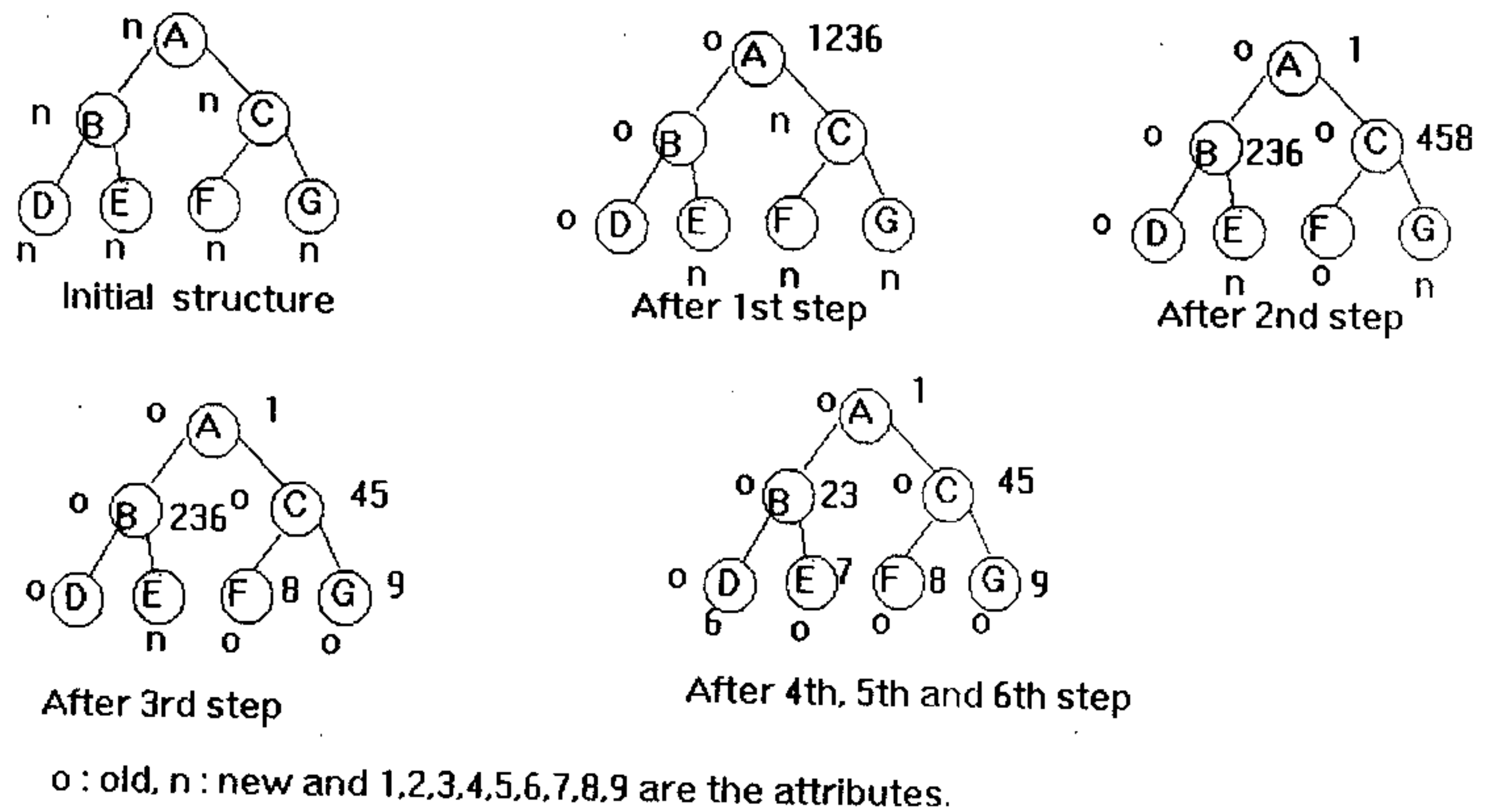


Figure 3.1

3.2 Verification

The verification algorithm will check three important aspects of the distribution of identifying attributes to the classes.

3.2.1 Completeness

Each attribute should be present in at least one class. It can be defined as the *completeness* criterion. Traversing once through the tree structure of classes and taking the union of the attributes present in those classes we can get the attributes which are present in at least one class. This set should be the same as the universal attribute set. If those sets are different, i.e. some of the attributes are still not included in any class description then we need more learning examples to run the learning algorithm again.

Also, we have to check that the learning instances should be such that after the learning algorithm no class should have the flag *new*. That is all the classes of the tree must be accessed at least once in the learning phase.

3.2.2 Uniqueness

All path expressions should be unique. From each leaf class we can get exactly one path upto the root as multiple inheritance is not allowed. The universal identifying attribute set being of cardinality n ,

we can consider about n bit binary numbers associated with each leaf class. This number is called *path identification number*. Let us order the n attributes as A_0, A_1, \dots, A_{n-1} . the union of all the attributes lying on the path from root to that leaf class. So corresponding to *total leaf attribute set* we get an n bit binary number, with 1 at the i th position if the attribute A_i is present and 0 if absent, which is the *path identification number*. We calculate the decimal integer corresponding to this binary number called *decimal path identification number*. Now we should check that *decimal path identification number* of any two classes should not be equal. If the learning examples are unambiguous then the learning example itself will take care of the uniqueness of each path. But if this uniqueness is found disturbed then we should go for a new set of learning examples.

3.2.3 Checking with examples

At the verification stage we can also take some more examples (which we may call as verification examples) targeting towards subtle changes if required. Those instances may not match with the given class behaviour. Let us consider the following example. Say, we have seven classes as follows:-
The root class C with attribute a_0 .

Two children classes C_1, C_2 of parent C . C_1 has attribute a_1, a_2 and C_2 has a_3, a_4 .

C_{11} child of C_1 with attribute a_5 .

C_{12} child of C_1 with attribute a_6 .

C_{21} child of C_2 with attribute a_7 .

C_{22} child of C_2 with attribute a_8 .

Now let us consider that one verification example is inserted present value for the attributes $\{a_0, a_1, a_2, a_5, a_8\}$ and its target class is C_{11} . So the verification algorithm will flag it as an example which is not matching exactly with the C_{11} and will propose the following alternatives for the domain expert:

1. Ignore the instance
2. Add a_8 as an attribute of C_{11}
3. Create one child C_{111} of C_{11} with attribute a_8
4. Create one child C_{111} of C_{11} with attribute a_8 and another C_{112} of C_{11} without any attribute.

Now it is very well understood that if most of the verification instances do not match with the classes they are targeted for, the class structure will need a substantial change from the initial structure. Also we have to take care of different types of mismatch. The change in class structure is always supervised by the domain expert. In our discussion we defer the changing of class structure to the *classification* stage where we will discuss the relevant algorithms.

3.3 Remarks

Let us consider, the number of classes = N_C , the number of leaf classes = N_L , the number of attributes = N_A , the number of examples = N_E and the maximum depth of the tree = D . In worst case for the learning algorithm we have to update the attribute set of each class in the tree for each instance which needs $O(N_C N_E)$ set operations. For completeness $O(N_C + N_L)$ time is required. For checking uniqueness we have to check all the paths from each leaf to the root which takes $O(N_L D)$ operations. Depending on the generated structure of the tree we will go for classification of the instances in the next chapter.

Chapter 4

Classification

This chapter discusses how an instance can be placed in a given schema after insertion. Here those instances are considered where each attribute is either present or absent (value 1 or 0) and there is no unavailable data. Firstly the instances are compared with leaf level classes for *exact match*. If it is not found, top-down search is done from the root for *approximate classification*. In this case matching may be possible only upto some intermediate class or a partially classified leaf and an *exception* may be flagged.

In this method a lot of *exceptions* may get accumulated in each class. If the number of such *exceptions* exceeds the application specific threshold value for that class and *matched* instances of the class becomes minority, the semantic purpose of existing class structure is defeated. Now the lattice needs to be restructured. However, such restructuring is usually done under the supervision of the domain expert.

4.1 Different matching concepts

This section discusses different matching procedures for the insertion of an instance to a class.

4.1.1 Path identification and exact match

We again talk about *path identification* concept considered in the previous chapter. The universal identifying attribute set being of cardinality n , we can consider n bit binary numbers associated with each leaf class. This number is named *path identification number*. Since multiple inheritance is not allowed, there exists only a single path from the root to the leaf node. Let us order the n attributes as A_0, A_1, \dots, A_{n-1} . Now we define *total leaf attribute set* attached to a leaf class as the union of all the attributes lying on the path from root to that leaf class. So corresponding to *total leaf attribute set* we get an n bit binary number, with 1 at the i th position if the attribute A_i is present and 0 if absent, which is the *path identification number*. We calculate the decimal integer corresponding to this binary number called *decimal path identification number*. We sort these numbers in ascending order. Since our assertion is that the *total leaf attribute set* of all the classes are different (as multiple inheritance is not allowed), The difference is also there for *path identification numbers* as well.

Now given an instance which contains only 0 and 1 values, we can generate a decimal number corresponding to that instance and use binary search to check whether it matches with any of the

decimal path identification numbers. If it matches then we can directly decide the membership of the instance. If it does not match with any one of the leaf classes then we have to start checking from the root for approximate classification. It is obvious that if there are L leaf classes then this operation needs $O(\log_2 L)$ comparisons. We call this *exact matching*. For *exact matching* with a leaf class C_L , the instance is obviously a member of all the classes on the path starting from root to C_L since classes other than C_L on that path are superclasses of C_L .

4.1.2 Perfect match

In this section we take care of those instances which fail to have *exact match* with any of the leaf classes. We call these instances as *exceptional instances* or in short *exceptions*. Now we will give the definition of *perfect match*. The definition will go by induction. We consider that all the instances coming in the schema are in the application domain, i.e. each of the instances will be a *perfect match* for the root class at depth 0. If any instance is not a *perfect match* with the root class then it does not belong to the application domain. So an instance is defined to be in *perfect match* with the root class if all the identifying attributes of the root class are present in that instance. Now an instance will be in *perfect match* with a class C at depth $i \geq 1$ if,

1. It is a *perfect match* with a class C_p at depth $i - 1$.
2. C_p is the parent of C .
3. All the identifying attributes of C are present in the instance.
4. Let $A' = \text{Union of all the identifying attributes of the classes which are children of } C_p \text{ except the attributes of } C$. For all the members of A' the instance has to contain a value 0.

The term *perfect match* differs from the *exact match* in the sense that a *perfect match* of an instance is its *exact match* with a class at an intermediate level of the class hierarchy and not at the leaf. So, the algorithm for approximate classification proceeds as follows:-

Input : An instance I which is not exactly classified.

Output: The class upto which perfect match is found.

Start Algorithm

$C_p = \text{The root class}; i = 0;$

if I is not a *perfect match* for C_p then report that

I does not belong to the database and exit;

flag = true;

while (flag)

 flag = false;

$i = i + 1;$

 if there exists a *perfect match* with a class C among the children of C_p at i th level then

 flag = true;

$C_p = C;$

 Mark the attributes which are matched;

 Endif

if there does not exist a *perfect match* then report the classification upto C_p and record the instance as an *exception* of C_p .

End Algorithm

So we can classify an instance upto some depth in the tree. We call it *under classification*. We keep an *under classification tag* with the class which contains at least one such *exception*.

4.2 Break

The purpose of this section is to present a partitioning mechanism of a class belonging to the hierarchy. The class hierarchy considered here is dynamic because the internal structure of the classes and the interrelationship among them may keep on changing as new instances are inserted to the system. i.e. one class can be broken into several classes. Thus, a class initially defined by the domain expert may be broken into several classes to equip the system for subtler classification of instances. The problem generated is mapped to graph theoretic domain and deterministic polynomial time algorithm is proposed for this purpose.

4.2.1 Restricted vertex connectivity

Let $G(V_1 \cup V_2, E)$ be an undirected bipartite graph and let $a, b \in V_1$ be such that there does not exist any vertex $c \in V_2$ so that both $(a, c), (b, c) \in E$ simultaneously. We define set $S_R \subseteq V_1 - \{a, b\}$ as a *restricted vertex separator* if every path from a to b passes through at least one vertex of S_R , i.e. a and b belong to different connected components of $G - S_R$. The minimum cardinality of any *restricted (a, b) vertex separator* is denoted by $N_R(a, b)$.

From G we construct a new graph $G' = (V_1, E')$ where $E' = \{(a, b); a, b \in V_1, \text{ and } (a, c), (b, c) \in E\}$

Lemma: If S_R is a minimum cardinality *restricted vertex separator set* of $G(V_1 \cup V_2, E)$, it is also a minimum cardinality *vertex separator set* of $G'(V_1, E')$ for $a, b \in V_1$ and vice versa.

Proof: We will prove the lemma by contradiction.

Let us consider removal of S_R does not separate a, b in $G'(V_1, E')$. Then there exists at least one path $p' = \{a = a_0, a_1, \dots, a_k = b\}$ in G' . Now, for each edge $(a_i, a_{i+1}) \in E'$, there exists some $v_i \in V_2$ such that $(a_i, v_i), (v_i, a_{i+1}) \in E$. Hence there exists a path $p = \{a = a_0, v_0, a_1, v_1, \dots, v_{k-1}, a_k = b\}$ in G . Hence we land in a contradiction.

To prove the other side let S be a minimum cardinality *vertex separator set* of $G'(V_1, E')$ for $a, b \in V_1$. Let the removal of S in $G(V_1 \cup V_2, E)$ does not separate a, b . Then after the removal of S there exists at least one path $p = \{a = a_0, v_0, a_1, v_1, \dots, v_{k-1}, a_k = b\}$ in G . Now from construction $p' = \{a = a_0, a_1, \dots, a_k = b\}$ is a path in G' even after removal, which contradicts the definition of S . (proved).

So, from the discussion in Chapter 2, we can calculate the *restricted vertex separator* between two vertices of a bipartite undirected graph in time $O(n^{\frac{1}{2}}e)$, where e is the number of edges in the transformed general undirected graph, i.e., $e = |E'|$.

The *restricted vertex connectivity* of a bipartite graph $G(V_1 \cup V_2, E)$, where we remove the vertices from V_1 is

$$c_R = \begin{cases} |V_1| - 1 & \text{if } G \text{ is complete bipartite} \\ \min\{N_R(a, b); a, b \in V_1, c \in V_2, \text{simultaneously}(a, c), (b, c) \notin E\} & \text{otherwise} \end{cases}$$

Calculation of c_R also takes polynomial time as it is equal in order to find the *vertex connectivity* of the transformed general undirected graph.

4.2.2 Problem Description

A class C consists of a set of identifying attributes A . An instance is defined by an *instance identification* and value 0 or 1 corresponding to each attribute depending on whether the attribute is absent or present respectively for the instance. As example, let us consider a class with attributes a_1, a_2, a_3, a_4, a_5 in the given order and some instances with instance id's $i_1, i_2, i_3, i_4, i_5, i_6$ described as follows :-

$$i_1 = 11100$$

$$i_2 = 11100$$

$$i_3 = 10011$$

$$i_4 = 10011$$

$$i_5 = 11111$$

$$i_6 = 10011$$

To clarify the concept, attributes a_1, a_2, a_3 are present but attributes a_4, a_5 are absent in id i_1 , again all the attributes are present in id i_5 . Only i_5 is a *perfect match* for the class C , others will face *exception* at the parent class of C .

Now it is interesting to watch that if we do not consider the instance i_5 , the situation is as follows:-

$$i_1 = 11100$$

$$i_2 = 11100$$

$$i_3 = 10011$$

$$i_4 = 10011$$

$$i_6 = 10011$$

So, we can now consider three classes C_1 with attribute a_1 , C_2 with attributes a_2, a_3 and C_3 with a_4, a_5 . We find that all the instances are members of class C_1 , i_1, i_2 are members of C_2 and i_3, i_4, i_6 are of C_3 .

This gives an idea to break a class C into three classes which offers better classification of the instances considered. Here C_2 and C_3 are the children of class C_1 .

A very natural question is that whether all the instances will be of some well structured form so that a class can be divided perfectly. The answer is obviously no. Because if we consider i_5 then the class C ceases to *break*. Now if we break the class C into C_1, C_2, C_3 then the instance i_5 will be an *exception* to class C_1 because after getting a *perfect match* upto class C_1 it will not get the same in the lower level. So in case we go for breaking a class in this way we have to minimize the number

of *exceptions* generated. It is interesting to note that in the example all the *exceptional instances* become *perfectly matched* and the *perfectly matched* instance i_5 is turned to an *exception* after *breaking*. So now the exact problem is to decompose one class into several classes (at least three, two children along with the parent class) by creating minimum number of *exceptions*.

First we identify those attributes which are present in all the instances of the class (as the a_1 attribute in the example). We construct a parent class with these attributes and all the instances of the original class will be the members of this class. If there is no such attribute exists, then there will not be any parent class. The class with remaining attributes and all the instances of the original class is the candidate for *break*, and let us define it as *residual class*. The remaining attributes of *residual class* are to be divided to more than one classes with generation of minimum number of *exceptional instances*. Now if a class C is considered for *break* in the class hierarchy, it must be a leaf class, otherwise multiple inheritance may occur. Let C_p = parent of C . Now after *breaking* the generated classes are C_1, C_2, C_3 say. If generated parent class C_1 exists, it will be the parent class of the children C_2, C_3 created through *break* and child class of C_p . But if C_1 does not exist, C_p will directly be the parent of the generated children classes C_2, C_3 . The graph theoretic modeling enables us to decide how the attributes and the corresponding instances will be distributed in the children classes.

4.2.3 Mapping

The residual class is mapped to the following undirected bipartite graph: $G = (I \cup A, E)$, where A = the attribute set of the residual class.

I = the instance set of the residual class.

$E = \{(a, i); a \in A, i \in I \text{ and instance } i \text{ has attribute } a \text{ present in it}\}$. (a, i) is unordered pair and hence the graph G is undirected and it is also bipartite.

So the problem of decomposing one class into several classes by creating minimum number of *exceptional instances* reduces to the following graph algorithmic problem :

Given an undirected graph $G = (I \cup A, E)$, A and I being the bipartitioning, to find the minimum cardinality set S_R , such that S_R is a subset of I and removal of vertices of S_R makes the graph disconnected. So we land into the problem of *restricted vertex connectivity* in an undirected bipartite graph and it can be done in polynomial time deterministically. The instances of S_R will be the *exceptions* of the generated parent class and the two disconnected components of the graph will construct two children classes. So if the disconnected components are $G_1 = (I_1 \cup A_1, E_1)$ and $G_2 = (I_2 \cup A_2, E_2)$, then one of the children classes will contain the attribute set A_1 and instances I_1 , the other A_2 and I_2 .

So if we consider the example in the previous subsection we find that $I = \{i_1, i_2, i_3, i_4, i_5, i_6\}$, $A = \{a_1, a_2, a_3, a_4, a_5\}$. After the *break* algorithm the parent class will have instance set $\{i_1, i_2, i_3, i_4, i_5, i_6\}$ and attribute set $\{a_1\}$. For the two children classes those will be $\{i_1, i_2\}$, $\{a_2, a_3\}$ and $\{i_3, i_4, i_6\}$, $\{a_4, a_5\}$. For the parent class i_5 will be an *exception* as it will not be a *perfect match* with any of its children classes.

4.3 Fuse

In this section we will consider *fuse* of more than one classes which are children of same parent. It will change the parent-child relationship of some classes, which basically changes the semantic structure of the tree. The syntactic structure of each path of the tree remains the same. The possibility of *fuse* will be reported to the supervisor who has the sole discretion of accepting or rejecting the operation depending on the semantic is-a relationship among the classes at different levels.

Let us consider we have two children classes C_1 and C_2 of same parent C . Let C_1 has attributes a_1, a_2 and C_2 has attributes a_1, a_2, a_3, a_4 . Such a situation may arise when *Add* is executed to create new class for classifying some of the *exceptions*. So we find that from the consideration of attribute set $C_1 \subset C_2$. Let the children of C_1 are C_{11}, \dots, C_{1n} and those of C_2 are C_{21}, \dots, C_{2m} . Now we do the following thing:-

1. Remove the parent-child relationship between C and C_2 .
2. Remove the attributes a_1, a_2 from C_2 .
3. C_2 becomes a child of C_1 along with C_{11}, \dots, C_{1n} .

So we find that all the *path identification numbers* of all the leaf classes remain undisturbed due to this operation, keeping the syntactic structure of each path same.

The relationships among the children classes of the same parent are to be found. In worst case it takes $O(n^2)$ set operations for this purpose. But since the subset relationship is transitive we may reduce some operations in actual case. For three classes A, B, C , if $A \subset B$ and $B \subset C$ then $A \subset C$. So if we keep the relationship between A and B , and between B and C then we don't have to go for any set operation between A and C . So some set operations can be saved keeping a table of set relations.

4.3.1 Role of Domain Expert

In the example of the previous subsection we may get another class C_3 which has the attributes a_3, a_4 . so we get $C_1 \subset C_2$ and also $C_3 \subset C_2$. So, it is undecidable which relation should be taken for restructuring. So here the discretion of the domain expert comes. We may consider the example of an archaeological database. It may have two classes; one idols of human figure(C_H) and the other of elephant (C_E). Now the identifying attributes of C_H are *two hands and two legs* and those of C_E are *trunk and tail*. Let the parent of C_H and C_E be C . Now if we find many instances of *Ganesh* with all the four attributes present, a new class C_G may be generated in *add* procedure. Now C_H, C_G or C_E, C_G can be *fused* from the attribute consideration, but from the domain knowledge it is clear that the class of idols of *Ganesh* C_G should not be *fused* with C_H or C_E and the domain expert should prevent the system from such *fusion*. Similarly the class *Man* may have a set of identifying features which is the subset of the set of identifying features present for the class *Centaur*. Same situation may occur for the *Horse*. However, *Centaur* can not be a child class of either *Horse* or *Man*.

4.4 Add

If a class accumulates very large number of *exceptions* which crosses the supervisor defined threshold, then restructuring of class hierarchy may be required. New classes may be generated adding new

branch to the class hierarchy for this purpose.

Let us consider two children classes C_1, C_2 of parent C . Let C_1 has the attributes a_1, a_2 and C_2 has a_3, a_4 . Now if number of instances come with all the attributes a_1, a_2, a_3, a_4 present and *perfectly matched* upto the class C , all those will become *exception* to class C . Now if there were some class C_3 with attributes a_1, a_2, a_3, a_4 as a child of C , then these *exceptions* would not be generated. So if the number of *exceptions* exceeds the prescribed limit in a class C , we have to create at least one new child class of C so that the number of *exceptions* becomes less than the threshold value for C . So it is better to generate a single class first such that the number of *exceptions* decrease maximally. So we have to find the pattern of attributes which generates the maximum cardinality. We push the part of *exceptions* of C , which will not remain its *exception* after creation of C_3 , to C_3 .

So to generalize let us consider a class C with children C_1, C_2, \dots, C_n with attribute sets A_1, A_2, \dots, A_n respectively. Let $A = \cup A_i \forall i$. C has *exceptions* above the threshold value. We have to find $S \subseteq A$ such that there exists maximum number of *exceptional* instances at class C having present value for the attributes in S and absent value for all the attributes of $A - S$. We will construct a class C_{n+1} with attribute set S and add it as a child of C along with C_1, C_2, \dots, C_n . Also if generation of class C_{n+1} gives any subset type of relationship with its peers, *fuse* algorithm may be used under the supervision of the expert. We declare class C_{n+1} as a *partially classified leaf*.

First we order the attributes of A as a_1, a_2, \dots, a_m . So corresponding to each *exceptional pattern* we will get an m -bit binary vector. Each vector will correspond to a decimal number. So only one scan through the *exceptions* will give the counts against each vector type and the time required will be linear.

4.4.1 Partially Classified Leaf

The class created by *add* operation (C_N say) does not have any child class. So C_N is a leaf. But the class is not properly classified in the sense that the instances coming to this class from its parent class may contain many other attributes except the attributes which define C_N . Now considering the instances in C_N , it may be possible to find certain pattern for *breaking* the class. The *partially classified* class is not added to the list of the leaf classes for *exact matching*. Now after *break* we will get classes at one level lower to C_N . These classes will then be considered *partially classified* and may be the candidates for *break* again. C_N will then become intermediate. It is the responsibility of the domain expert to decide how long the recursive *break* will continue. After that it is also the duty of the expert to declare some of the *partially classified* leaves as leaf classes and insert them in the leaf class list so that they can be considered for *exact matching* later.

4.5 Tree Restructuring

After the application of the algorithm for approximate classification, If number of *exceptions* exceeds threshold for the *underclassified class*, we go for the following steps. All the steps are under the control of a domain expert, i.e. the expert can accept or reject any of the steps considering the application domain.

1. *Add* a new child class C_N of the underclassified class C .
2. Push the part of *exceptions* of C , which will not remain its *exception* after creation of class C_N , to C_N .
3. *Break* C_N recursively if possible.
4. Explore the *Fuse* possibilities of C_N with any of its peer class and restructuring the hierarchy.
5. Transform some partially classified leaves to leaf level classes.

4.6 Example

In this section we consider a class structure and some restructuring on it depending on the input instances. We consider the attribute set $A = \{a_0, a_1, a_2, a_3, a_4, a_5, a_6, b_0, b_1, b_2, b_3, b_4, b_5, b_6\}$. We have seven classes in the hierarchy $C_0, C_1, C_2, C_3, C_4, C_5, C_6$ and C_7 as shown in the figure. Now let us consider we got *exact match* for 500 instances initially, 100 for each of the leaf class. Then we found 10 instances with attribute set $\{a_0, b_0, b_1, b_3, a_3, a_4\}$, 100 instances with $\{a_0, b_0, b_1, b_2, b_3, b_4, a_3, a_4\}$, 100 instances with $\{a_0, b_0, b_1, b_2, b_3, b_4, a_5, a_6\}$, and 10 instances with $\{a_0, b_0, b_1, b_2, b_3, b_4, a_3, a_5\}$. So we get all these 220 instances will be *exception* to the class C_4 . Initially we got 300 *exact matches* for the classes C_5, C_6 and C_7 , which are all children of C_4 . Let us consider the 220 *exceptions* cross the threshold value for the class C_4 . So we need to create a new class so that the number of *exceptions* in the class C_4 decreases. The union of attribute sets of C_5, C_6 and C_7 is $\{b_1, b_2, b_3, b_4, b_5, b_6\}$. Now we find that b_1, b_2, b_3 and b_4 are those attributes which are there in 210 *exceptional instances*. So we form a class C_8 with those attributes and corresponding 210 instances. The 10 instances with attribute set $\{a_0, b_0, b_1, b_3, a_3, a_4\}$ still remain *exceptions* to C_4 after the creation of C_8 . Now if we run *break* on the class C_8 we find that two children classes C_9 and C_{10} are generated under C_8 with attribute sets $\{a_3, a_4\}$ and $\{a_5, a_6\}$ respectively with 100 instances in each, and 10 attributes stays in the class C_8 as *exceptions*. Again *fuse* may be applied either between

C_5, C_8 or C_6, C_8 under the supervision of the domain expert.

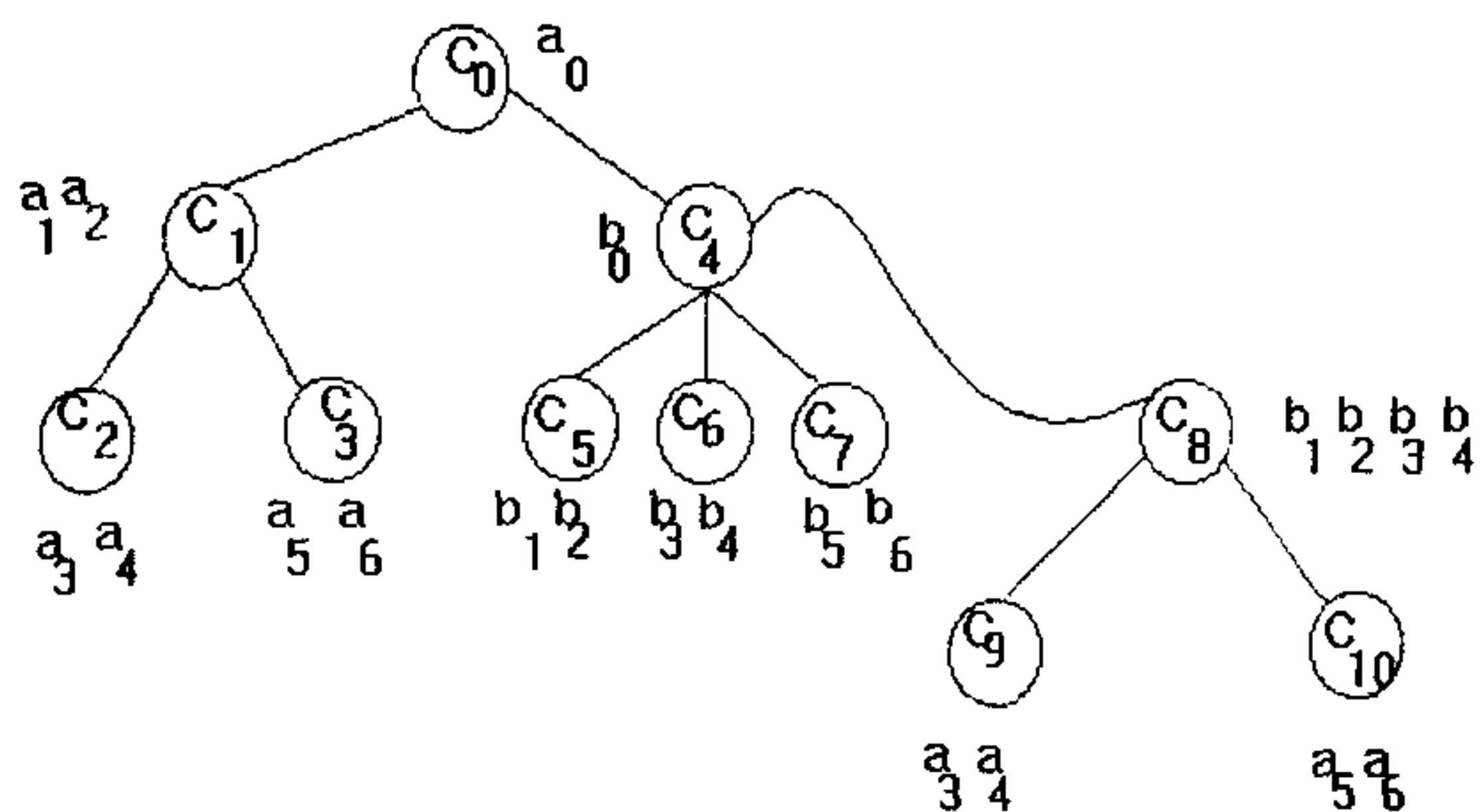


Figure 4.1

4.7 Remarks

As claimed earlier, this chapter reveals that the schema is really dynamic. However, since the learning is supervised, any mistake on the part of the domain expert may cause an erroneous augmentation of the class structure.

Chapter 5

Classification with some of the attribute values unknown

When some attribute values are unavailable (value marked as X) for some instances, we try for a match ignoring the attributes with X values. If a match at leaf level fails we go for a top-down search either ignoring the X valued attributes or assigning 1 or 0 in place of X and try for a maximal match, i.e. a *perfect match* to an intermediate class at the maximum depth possible.

5.1 Forced match

If an instance contains some X values, one way to classify the instance is to neglect the X values. For each leaf class we compare the attributes of the instance which do not have X values with those of the leaf class. If there is no mismatch with the 0 or 1 values at the corresponding attributes then such an *exact match* is called a *forced match*.

Three cases may occur:

1. There exists only a single class at leaf level which is a *forced match* with the instance. Here we classify the instance in that class with a forced flag.
2. There exists more than one classes at leaf level which offer *forced match* with the instance. Here we go for *maximum-1 match*. Here, that class is chosen as most suitable where maximum number of 1-valued attributes (i.e. present) match with the instance. Even then also if we get more than one classes, we can report the list of all those classes for supervisor's choice. If supervisor fails to give a decision we name this situation as *multiple forced match*. Top-down search is applied in this event.
3. There does not exist any class which is a *forced match* with the instance. We call this *failed forced match*. In this case top-down search should be applied.

As example of case 1, let us consider 9 attributes a_0 to a_8 for a schema as shown in the figure. Let one leaf class C_3 has total leaf attribute set $\{a_0, a_1, a_2, a_5\}$. An instance I has 1 value at a_0, a_1, a_5 ,

X value at a_2, a_7 , and 0 value at a_3, a_4, a_6, a_8 . If we do not consider the attributes a_2 and a_7 in matching, for other attributes the instance I matches with C_3 . So I is a *forced match* with C_3 . For the case 2 if we consider an instance with 1 value of a_0, a_3, a_4 , X value at a_7, a_8 , and 0 value for other attributes then it will be *forced match* for both the classes C_5 and C_6 , and it needs domain expert's interference to break the tie. To consider the third case if we consider an instance with 1 value at a_0, a_1, a_5, a_7 , X value at a_2 , and 0 value at other attributes, it can not be *forced matched* with any of the leaf classes and top-down approach should be applied.

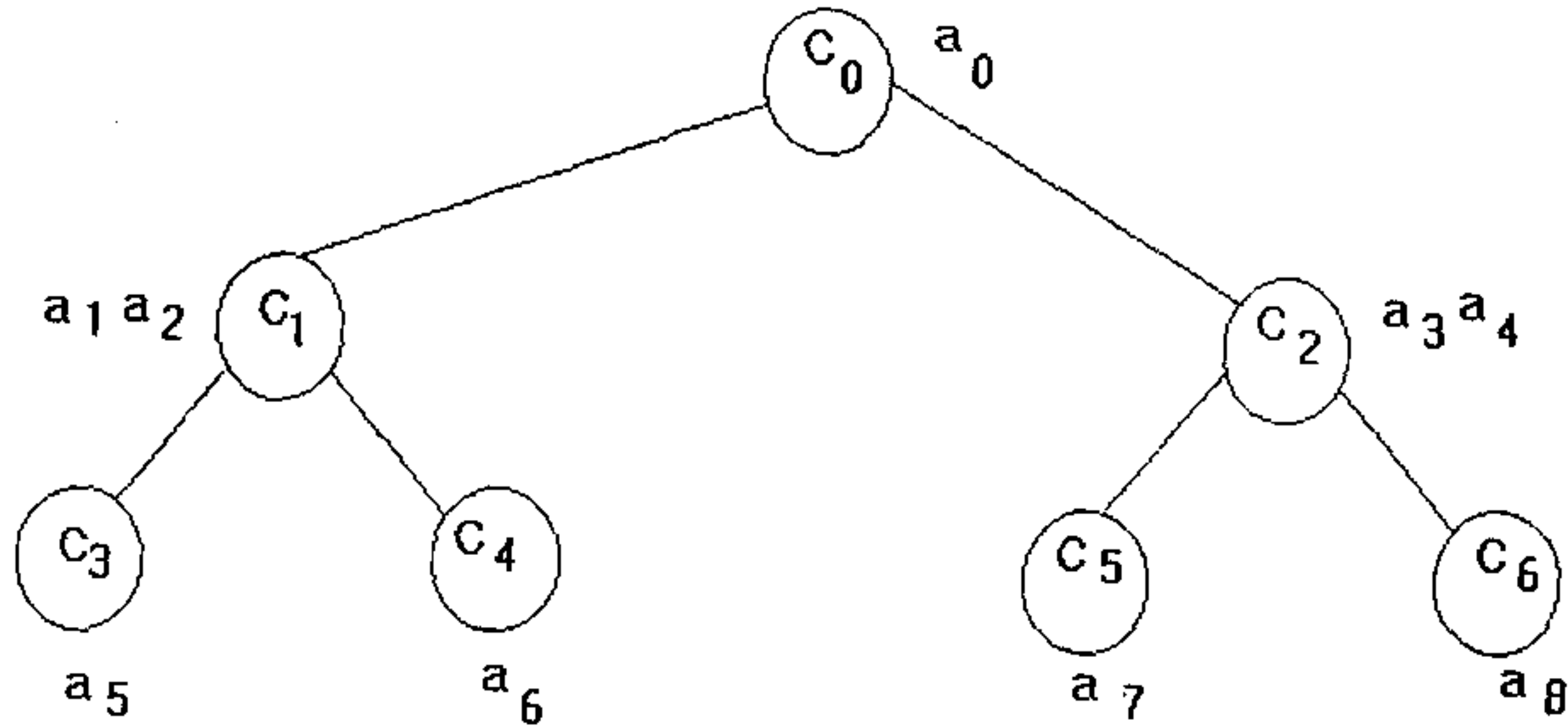


Figure 5.1

5.2 Best match

Here we define *best match*. The definition goes more or less in the line of *perfect match*. An attribute I will be a *best match* with root class if all the attributes of root class are present or unknown in that instance. An instance will be a *best match* with a class C at depth $i \geq 1$ if,

1. It is *best match* with a class C_p at depth $i - 1$.
2. C_p is the parent of C .
3. All the identifying attributes of C are 1 or X in the instance.
4. Let $A' =$ Union of all the identifying attributes of the classes which are children of C_p except the attributes of C . For all the members of A' the instance has to contain a value 0 or X .

5. If there exists more than one classes among the children of C_p which satisfy the first four conditions then try to break the tie selecting that class C which has maximum number of 1 match with the instance. If that also provides more than one class, we have to go for supervisor's help to break the tie.

It should be noted at this point that considering the application domain we can go for assigning weight to the attributes. In that event we do consider *maximum weighted matching* which minimizes the supervisor's interference.

The algorithm for approximate classification in presence of some unavailable attributes proceeds as follows:

Input : An instance I which is not exactly classified.
Output: The class upto which *best match* is found.
Start Algorithm
 C_p = The root class; $i = 0$;
if I is not a *best match* for C_p then report that
 I does not belong to the database and exit;
flag = true;
while (flag)
 flag = false;
 $i = i + 1$;
 if there exists a *best match* with a class C among
 the children of C_p at i th level then
 flag = true;
 $C_p = C$;
 Mark the attributes which are matched;
 Endif
 if there does not exist a *best match* then report the
 classification upto C_p and record the instance as an
 exception of C_p .

End Algorithm

Hence with the concept of *best match* we can classify an instance upto some depth in the tree. This is also considered as *under classification* for which a *tag* is attached with the class which contains at least one such *exception*.

5.3 Tree Restructuring

The tree restructuring is a little bit different from the concept of the previous chapter though the basic steps are same. After running the algorithm for approximate classification, if number of *exceptions* exceeds threshold for the *underclassified class*, we go for the following steps. As in the previous chapter all the steps are under the control of a domain expert, i.e. the expert can accept or reject any of the steps considering the application. But here the expert has some more information to supply. We write the same steps written in the previous chapter along with the additional requirements.

1. *Add* a new child class C_N of the underclassified class C . { For the *add* algorithm the X 's should be assigned either 0 or 1. So, we have to list the attributes which had X value in any of the *exceptional* instances. For those attributes before the invocation of *add* algorithm the expert should assign 0 or 1 value. If that is not explicitly decided, then for k such attributes 2^k possible combinations should be checked, which is very large for large k , and could not be used for practical purposes. In that event the system may once consider all the X values to be 0, and then 1 for another case and report the domain expert both the outputs. Though the 0-1 assignment will be used for running the algorithm, the instances will contain the same X values in the system for the corresponding attributes. }
2. Push the part of *exceptions* of C , which will not remain its *exception* after creation of class C_N , to C_N considering the assignments in step 1.
3. *Break* C_N recursively if possible. { Here also we have to take care of the same consideration as in 1. }
4. Explore the *Fuse* possibilities of C_N with any of its peer class and restructure the hierarchy.
5. Transform some partially classified leaves to leaf level classes.

5.3.1 Breaking of leaf level class

Now we consider another concept of tree restructuring apart from the previous points. In a leaf level class many of the instances may be *forced matched*. The leaf class inherits number of attributes from the parent classes upto the root and it also have some of the attributes for itself. Now for the leaf class(C) attributes we can construct an attribute versus instance table. If we check the parent class C_p of C , we may find some *exceptions* which are created only due to the class C . The concept is as follows:-

Let A_i be the attribute sets of classes C_i which are the children of parent class C_p . So, C is C_k for some attribute set A_k . Let $A = \cup A_i \forall i - A_k$.

Now among the *exceptions* of class C_p , there may be instances which contain 0 value for all the attributes of set A and 1 value for at least one of the attributes of set A_k . Those instances may not remain as *exceptions* to class C_p if the structure of the class C_k changes. We also add these instances in the table. So we get an attribute versus instance table where the attributes are from attribute set A_k , and the instances are the instances of C_k and some of the *exceptions* of C_p . Now the entry of the elements of the table will be 0, 1, X . Now assigning proper 0 or 1 values in place of X as discussed in the step 1 of tree restructuring we can apply the *break* algorithm to the class C_k . If the number of *exception* generated after the *break* is less than the threshold value, the expert may accept the *breaking* of class C_k , and in that event we will get two children classes of C_k . Hence we will get better specialization in the structure. In that event C_k will get out of the leaf class list, and two new classes will enter there with proper *path identification* values.

5.4 Remarks

In this chapter we have discussed tree restructuring with some of the attribute values unknown for some of the instances. The application of 0 or 1 in place of X extensively depends on the application domain and knowledge of the expert. Dependency among the attributes may also play some important role which takes care of interpretation of a set of attribute values in response to assignment of another set of attribute values in the schema.

Chapter 6

Application and Future possibilities

6.1 Database Applications

In the preceding sections we discussed different techniques to organize and maintain database classes. The algorithms help to get database classes that more accurately reflect the real classification among the attributes and instances.

1. **Schema Generation and Evolution.** The techniques described in the previous sections are applicable to schema generation and maintenance and can be used to help the database designers. Specifically the hierarchical structure of classes gives a direct resemblance with Object Oriented Schema. The system being incremental, new classes can be added to the schema at any time along with the instances. The process is open ended, leading to more complex and accurate schema as more information is added to it.
2. **Schema Integration.** Given different hierarchical structures of classes for different domain of applications, we can generate a global dummy root class, and the root classes of all the different applications will be the children of that. The union of universal attribute sets considered for all the schemas will be considered as the universal attribute set of the integrated schema. *Synonyms* and *homonyms* should be properly renamed to avoid conflicts.
3. **Query Processing.** In the proposed system, each instance has instance identification number. Hence for any query regarding an instance we can easily find whether the instance is properly classified or not, and the classes where the instance is present. Again a query asking for the list of instances with some specific feature combinations can be answered considering the union, intersection or some other set operations on the attribute set of the present classes. For this type of query, exact match with some specific class at leaf level may not be found. In that event *exact match* may be found upto an intermediate class and such incomplete class specification will be presented.
4. **Automatic Generation of Views.** If we consider universal table using the universal attribute set of the schema, the classes generated will work as views on that table. The augmentation of the class structures would also alter such views and new views may also be introduced.

6.2 Future Work Direction

1. We have considered the attribute values to be 0 and 1 only. Extending the work towards general quantitative and qualitative value domain will be an interesting field of study.
2. For the classification with some of the attribute values unknown(X) we have either neglected them or expected that the domain expert will assign 0 or 1 value for those. One possibility is to consider the real space $[0, 1]$ and to assign application specific fractional values to unknown attributes and try for some new classification algorithm. This type of algorithm will be able to solve a considerable amount of item 1.
3. Some coding techniques may be used for approximate classification. Concept like *Hamming Distance* may be explored for this purpose.
4. Assignment of weight for each attribute and dependency among them may reduce the interference of the domain expert. For any application specific design of this type of evolutionary schema these considerations may produce more accurate classification.

6.3 Conclusion

As classification is a general problem, the description of the problem always becomes application specific. We have tried to state the problem and its solution mostly in application independent way so that any application can be developed over this skeleton. We feel the system can successfully be used as an intelligent front end engine to any database.

References

1. Henry F. Korth and Abraham Silberschatz, *Database System Concepts*, Second Edition, McGraw-Hill, Inc., 1991.
2. Elmasari and Navathe, *Fundamentals of Database Systems*, Second Edition, The Benjamin Cummings Publishing Company, Inc., 1994.
3. Heikki Mannila and Kari-Juho Raiha, *The design of Relational Databases*, Addison-Wesley Publishing Company, 1994.
4. Alfons Kemper and Guido Moerkotte, *Object Oriented Database Management: Application in Engineering and Computer Science*, Prentice Hall, 1994.
5. Kurt Mehlhorn, *Graph Algorithms and NP-Completeness*, Monographs on Theoretical Computer Science", Springer-Verlag, 1984.
6. Timothy W. Finin, Charles K. Nicholas, Yelena Yesha (Eds.), *Information and Knowledge Management*, Lecture Notes in Computer Science (752), Springer-Verlag, 1993.
7. Howard W. Beck, Tarek Anwar and Shamkant B. Navate, *A Conceptual Clustering Algorithm for Database Schema Design*, IEEE Transactions on Knowledge and Data Engineering, pp. 396-411, vol. 6, no. 3, June 1994.
8. Paul E. Utgoff, *Incremental Induction of Decision Trees*, Machine Learning, 4, pp. 161-186, 1989.
9. Yi Deng and Shi-Kuo Chang, *A G-Net Model for Knowledge Representation and Reasoning*, IEEE Transactions on Knowledge and Data Engineering, pp. 295-310, vol. 2, no. 3, September 1990.
10. Padhraic Smyth and Rodney M. Goodman, *An Information Theoretic Approach to Rule Induction from Databases*, IEEE Transactions on Knowledge and Data Engineering, pp. 301-316, vol. 4, no. 4, August 1992.
11. Ramin Yasdi, *Learning Classification Rules from Database in the context of Knowledge Acquisition and Representation*, IEEE Transactions on Knowledge and Data Engineering, pp. 293-306, vol. 3, no. 3, September 1991.
12. M. K. Crowe, *Object Systems over Relational Databases*, Information and Software Technology, pp. 449-461, vol. 35, no. 8, August 1993.
13. Eric N. Hauson, Tina M. Harvey and Mark A. Roth, *Experiences in Database System Implementation Using a Persistent Programming Language*, Software - Practice and Experience, vol. 23(12), pp. 1361-1377, December 1993.
14. Elaine Rich, *Artificial Intelligence*, McGraw-Hill, 1986.
15. Robert J. Schalkoff, *Artificial Intelligence : An Engineering Approach*, McGraw-Hill, 1990.