Algorithm for defuzzification of multi-valued taxonomic attributes in similarity-based fuzzy relational databases

M. Shahriar Hossain, Rafal A. Angryk

Department of Computer Science
Montana State University
Bozeman, MT 59717-3880, USA
{mshossain, angryk}@cs.montana.edu

Abstract. In this work we investigate our potential ability to discover knowledge from multi-valued attributes (often referred in literature on fuzzy databases as fuzzy collections [1-3]), that have been utilized in fuzzy relational database models [4-7] as a convenient way to represent uncertainty about the data registered in the data tables. We present here implementation details and extended tests of a heuristic algorithm, which we used in the past [8-11] to interpret non-atomic values stored in a fuzzy relational database. In our evaluation we consider different data imprecision levels, as well as diverse shapes of similarity hierarchies.

Keywords: fuzzy relational databases, fuzzy collections, non-atomic symbolic values.

1 Introduction

The dynamic growth of data mining research and applications [12] carries significant importance for researchers working in areas related to fuzzy databases. Successful future of unconventional database models becomes more and more dependent on the development of consistent and time-efficient mechanisms for mining the data such databases are capable to store and process.

Fuzzy databases let the user reflect uncertainty about inserted information via the insertion of multiple descriptors in every column of the data table. At the same time, majority of currently available data mining algorithms, allows the user to deal only with atomic attribute values. This leads to the challenge of developing a mechanism allowing consistent interpretation of the non-atomic values stored in the fuzzy databases. This process can be interpreted as mapping (i.e. defuzzification) of fuzzy tuples from tables in fuzzy relational databases into a set of atomic records, that are compatible with 1NF definition and can be analyzed using regular data mining techniques.

In this work we will focus on problem of interpretation of multi-valued, nonnumeric entries in the similarity-based fuzzy relational databases. We have decided to chose this topic as these types of entries appear in large number of applications, starting from multiple online surveys about customers' preferences (e.g. *mark types of food you like*), and finishing on reports from medical examinations (e.g. *identify areas of pain, describe its severity, etc.*). In this paper we will present a heuristic allowing to transfer non-atomic symbolic values to the singleton forms, that can be then interpreted by many regular data mining algorithms. At the very beginning of this article, however, we want to emphasize that the point of this paper is not to argue about accuracy or efficiency of the approach that has been used, but rather to raise awareness of the problem, and to show that a defuzzification of uncertain data is achievable.

In the next section, we provide a brief review of the necessary background that covers the following topics: (1) fuzzy database model incorporating usage of non-atomic values to reflect uncertainty, (2) taxonomic symbolic variables and some of their properties, (3) our heuristic to interpret non-atomic entries in the fuzzy tuples. In the section 3, we present implementation details of our approach, discuss the artificial data sets we created for testing, and finally discuss results we generated. Finally, in section 4, we briefly summarize conclusions coming from our investigation and point out new directions of future research.

2 Background

2.1. Fuzzy Database Model

There are two fundamental properties of fuzzy relational databases, proposed originally by Buckles and Petry [4-5] and extended further by Shenoi and Melton [6-7]: (1) utilization of non-atomic attribute values to characterize features of recorded entities we are not sure of, and (2) ability of processing the data based on the domain-specific and expert-specified fuzzy relations applied in the place of traditional equivalence relations.

In fuzzy database model it is assumed that each of the attributes has its own fuzzy similarity table, which contains degrees of similarity between all values occurring for the particular attribute. Such tables can be also represented in the form of the similarity hierarchies, named by Zadeh [13] – partition trees that show how the particular values merge together as we decrease similarity level, denoted usually by α. Example of partition tree for the domain *Country*, and *Food-Type* is presented in the figures 1 and 2, respectively.

2.2. Taxonomic symbolic variables and their analysis.

In our opinion, [14] contains the most comprehensive collection of contemporary investigations on extraction of statistical information from symbolic data. Based on the classification presented in the book, the data type discussed in this paper should be characterized as *generalized taxonomic symbolic variables*. These types of data entries are expected to carry no quantitative meaning, but yet the values can be nonlinearly ordered in a form of rooted, hierarchical tree, called by the Diday [14] a *taxonomy*. Such variables are called taxonomic or tree-structured variables. In our opinion – Zadeh's partition trees [13], as presented in the figure 2, represent such taxonomies, allowing to incorporate fuzzy similarity relations not only to data querying, as it has been used in the past[], but also to the data analysis.

In [14], chapter six [15] has been devoted almost entirely to the problem of derivation of basic description statistic (in particular: medians, modes and histograms) from multi-valued symbolic data. The authors propose transformation of non-atomic values to collection of pairs in the format of (ξ, φ), where ξ stands for a singleton (i.e. atomic) symbolic value, and φ represents ξ's observed frequency. This is achieved by extending the classical definition of frequency distribution. For a multi-valued variable, the new definition states that the values taken into account the *observed frequency* can be positive real numbers, instead of positive integers, as is the case for a single-valued variable.

The observed frequency distribution of a multi-valued variable *Z*, for a single data point, is the list of all values in the finite domain of *Z*, together with the percentage of instances of each of the values in this data point.

In the middle of the chapter, the authors mention that other definition of frequency distributions can be proposed, suggesting taking into account the natural dependencies between the symbolic values, as reflected by the provided attribute's taxonomy. This observation gave us important motivation for the work presented below.

2.3. Similarity-driven Vote Distribution Method for interpretation of non-atomic values.

Following the motivation presented in the section above we attempted to develop a simple method to transfer non-atomic values in the fuzzy records to the collection of pairs including atomic descriptors and their fractional *observed frequencies*. In our wok, however, we wanted to utilize background knowledge about attributes' values, which is stored in fuzzy databases in form of fuzzy similarity relations [13]. In this section we present a simple example to introduce our approach.

We want a reader to assume for a moment that he/she needs to find a drugs' dealer who, as a not-confirmed report says (i.e. our fuzzy tuple), was recently seen in *{Canada, Colombia, Venezuela}*? The most trivial solution would be to split the count of observed frequency equally among all inserted descriptors, that is to interpret the entry as the following collection *{Canada|0.(3), Colombia|0.(3), Venezuela|0.(3)}*. This approach however does not take into consideration real life dependencies, which are

reflected not only in the number of inserted descriptors, but also in their similarity (represented by a taxonomy of values, reflecting a pre-defined fuzzy similarity relation).

In our work we used a simple heuristic [8-11] letting us to replace the even distribution of a vote with a nonlinear spread, dependent both on the similarity of inserted values and on their quantity. Using the partition tree built from the fuzzy similarity table (grey structure in Fig. 1), we can extract from the set of the originally inserted values those concepts which are more similar to each other than to the remaining descriptors. We call them *subsets of resemblances* (e.g. *{Colombia, Venezuela}* from the above example). Then we use them as a basis for calculating a distribution of a database record's fractions. An important aspect of this approach is extraction of the *subsets of resemblances* at the lowest possible level of their common occurrence, since the nested character of fuzzy similarity relation guarantees that above this α-level they are going to co-occur regularly.

Our heursitc algorithm is pretty straightforward. Given (1) a set of attribute values inserted as a description of particular entity, and (2) a hierarchical structure reflecting Zadeh's partition tree [13] for the particular attribute; we want to extract a table, which includes (a) the list of all subsets of resemblances from the given set of descriptors, and (b) the highest level of α-proximity of their common occurrence. We then use the list to fairly distribute fractions of the original fuzzy database record.

Our algorithm uses preorder recursive traversal for searching the partition tree. If any subset of the given set of descriptors occurs at the particular node of the concept hierarchy we store the values that were recognized as similar, and the adequate value of α. An example of such a search for subsets of resemblances in a tuple with the values *{Canada, Colombia, Venezuela}* is depicted in Fig. 1. Numbers on the links in the tree represent the order in which the particular subsets of similarities were extracted.

After extracting the *subsets of resemblances*, we apply a summarization of α values as a measure reflecting both the frequency of occurrence of the particular attribute values in the *subsets of resemblances*, as well as the abstraction level of these occurrences. Since during the search the country *Canada* was reported only twice, we assigned it a grade *1.4* (i.e. *1.0+0.4*). For *Colombia* we get: *Colombia|(1.0 + 0.8 + 0.4) = Colombia|2.2*, and for the last value: *Venezuela|(1.0 + 0.8 + 0.4) = Venezuela|2.2*.

At the very end we normalize grades assigned to each of the entered values: *Canada |(1.4/5.8) = Canada |0.24, Colombia |(2.2/5.8) = Colombia |0.38,Venezuela |(2.2/5.8) = Venezuela |0.38*. This leads to the new distribution of the record's fractions, which, in our opinion, more accurately reflects real life dependencies than a linear-split approach.

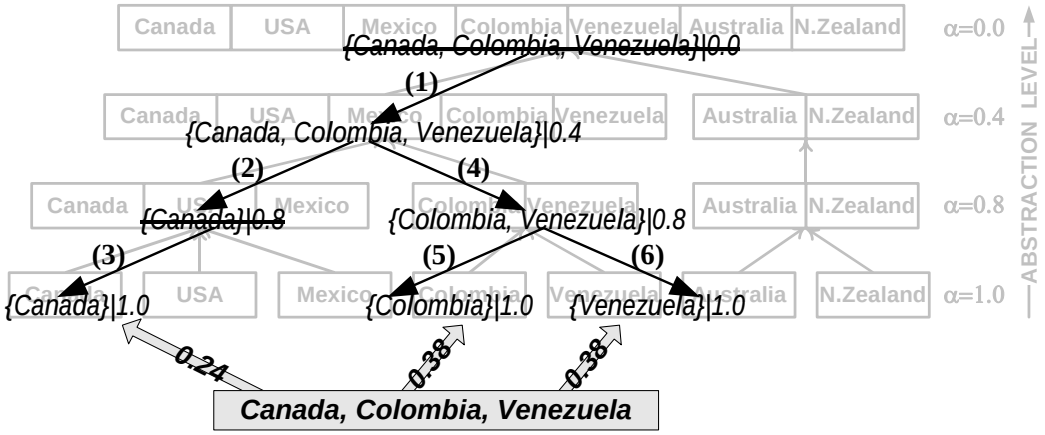

Fig. 1. Subsets of Resemblances extracted from the Partition tree.

3 Implementation of the defuzzification algorithm

The implementation of our defuzzification system is based on a class named `TreeNode`, which contains only a single node of the fuzzy similarity hierarchy, pointers to its immediate descendants, and some public methods to access them. Each descendant is an instance of the same class `TreeNode`. Hence, a `TreeNode` contains pointers to descendent `TreeNode`. Descendents that do not point to any other child node basically points to null `TreeNode` ($\{\Phi\}$). This is depicted in Fig. 2.

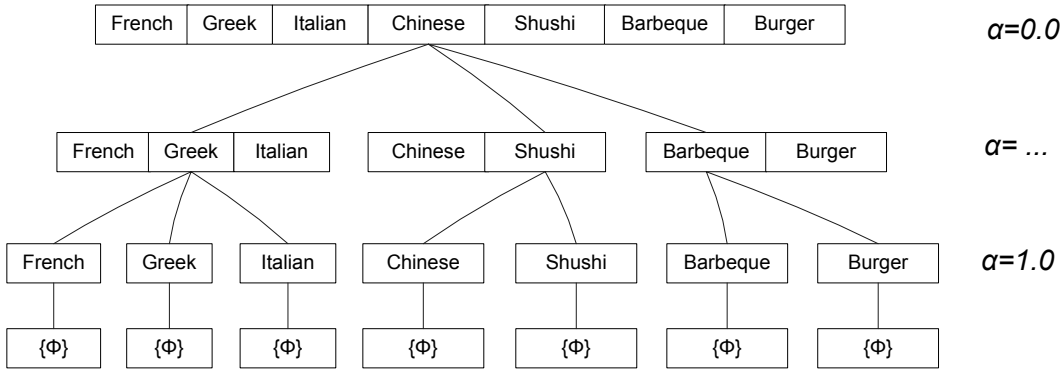

Fig. 2. Object representation of a partition tree.

The `TreeNode` class is given below for better illustration.

```
public class TreeNode {
  int childCount = 0;
  Vector nodeData = new Vector();
  Vector childrenNodes = new Vector();
  /* THIS IS NOT A BINARY TREE. SO IT CAN CONTAIN ANY NUMBER OF
     CHILDREN. childrenNodes CONTAINS ELEMENTS OF TYPE TreeNode */
  public TreeNode (Vector colon) {
    /* CREATES A NEW INSTANCE OF TreeNode*/
    nodeData = (Vector) colon.clone();
    childrenNodes = new Vector();  }
  public void addChild(TreeNode childNode){
    childrenNodes.add(childNode);
    childCount++;  }
  public int getTotalChildren(){
    /* e.g., IN THE CASE OF A NODE WITH THREE DESCENDENTS , THIS
    FUNCTION WOULD RETURN 3.*/
    return childCount;   }
  public Vector getNodeVector (){
    /* RETURNS THE PARENT NODE */
    return nodeData;   }
  public void setNodeAsLeaf (){
    childrenNodes.clear();  }
  public Vector getChildrenVector (){
  /* RETURNS THE CHILDREN NODES AS A VECTOR*/
    return childrenNodes;   }
  public TreeNode getChild (int i){
    /* RETURNS i-th CHILD WHERE EACH CHILD ITSELF IS A TreeNode*/
    return ( (TreeNode) childrenNodes.get(i) );   }
}//END OF public class TreeNode
```

The basic algorithm uses preorder recursive tree traversal (depth-first search, DFS) for searching matching subsets in the partition tree. Our goal is to find at each node of the partition tree the largest matching subset. The high level outline of the algorithm is portrayed in the following code. `List` and

`TreeNode` parameters of the `partitionTreeTraversal` function are passed by reference where other parameters are passed by value. The method `clone()` of an object returns entirely a new instance of the cloned object to avoid changes to a referenced parameter while the referenced parameter is necessary to be kept intact for the upper levels of recursion. A `Vector` can store a collection of entered attribute values, e.g. *{French, Greek, Italian}* and it can be used as an instance of `searchVector` of the algorithm. Besides, a `Vector` can also be a collection of other `Vectors`. On the other hand, a `List` is a collection of `Vectors` e.g., during the first call of `partitionTreeTraversal` the parameter `sList` should contain a sorted (descending order, based on the size of subsets) list of all the possible subsets of `searchVector` except the empty subset (*{Φ}*). The operation denoted by "–"in the algorithm below is considered a regular *SetDifference* operation.

```
public Vector partitionTreeTraversal(List sList, TreeNode masterNode,
                                     Vector searchVector, int level){
  List subsetList = sList.clone();
  int totalChildren = masterNode.getTotalChildren();
  Vector masterVector = masterNode.getNodeVector();
  Vector subsetVector = GET THE LONGEST SUBSET FROM subsetList THAT
                        IS FOUND IN THE ROOT OF masterNode;

  if (subsetVector.size()==0){
    return {Φ};
  }
  else if (subsetVector.size()==searchVector.size()){
    UPDATE THE CORRESPONDING ENTRY FOR EACH ELEMENT OF searchVector or
    subsetVector WITH CORRESPONDING α–VALUE OF level-TH LEVEL ;
  }
  else{ //IF PARTIALLY AVAILABLE
    ADD CORRESPONDING α-VALUE OF level-TH LEVEL TO THE CORRESPONDING
    ENTRY FOR EACH ENTITY OF subsetVector ;
  }

  Vector resultVector = searchVector.clone();
  for (int i=0; i<totalChildren; i++){
    Vector temp =
    partitionTreeTraversal( subsetList, masterNode.getChild(i),
                            subsetVector, level+1 );
    subsetList = subsetList – (ALL SUBSETS THAT HAVE AT LEAST ONE
                               ENTITY OF temp);
    resultVector = resultVector – temp;
    if (resultVector == {Φ} )     //NOTE:[1]
      break ;
  }
  if (totalChildren==0)    /*IF masterNode IS A LEAF*/
    return subsetVector;
} //END OF public void partitionTreeTraversal
```

3.1 Analysis of the Algorithm

Let us assume that the average branching factor of the partition tree is *b* and the number of abstraction levels in the tree is *d*. If each node has *b* descendents, then the root (*level 1*) has 1 node, *level 2* has *b* nodes, level 3 has b^{3-1} nodes, …., *level d* has b^{d-1} nodes. Hence the total number of nodes, *N* in the tree is: $N = \sum_{k=1}^{d} b^{k-1}$, leading to $Nb = \sum_{k=1}^{d} b^{k}$, where *k* reflects the current *level* of the *partitionTreeTraversal* algorithm. The last equation produces the following result:

[1] Do not traverse other branches because all entities are already found in the previously traversed branches.

$Nb - N = N(b-1) = b^d - 1$, thus $N = \frac{b^d - 1}{b-1}$. Therefore, in the worst case when the `searchVector` is dispersed in all the leaf nodes of the partition tree at d^{th} level, `partitionTreeTraversal` has to visit a total of $\frac{b^d - 1}{b-1}$ nodes which would result in a time complexity of $O(b^d)$ which should not be alarming despite of exponential growth, as typically $b<<N$ and $d<<N$ and usually $b<(d+1)$ for our experiments. The worst case search scenario occurs when the imprecise attribute of the fuzzy record, stored in `searchVector`, contains every element of the root node of the partition tree; in our example – if the `searchVector` is *{French, Greek, Italian, Chinese, Shushi, Barbeque, Burger}*. In such case the traversal would occur to every subtree of the hierarchy of Fig. 2. Such situation, however, should appear rather rarely as this type of entries is used in fuzzy relational database only if total lack of knowledge concerning an attribute value occurs, and in practice can be replaced by leaving the record with no value (i.e. NULL).

Obviously, the best case is when the entire `searchVector` is always found in the first descendent of every node along the traversal trail starting from the root. This case is possible only when the `searchVector` is atomic (as for example, *{French}*, considering the partition tree of Fig. 2) causing a traversal of a total of d nodes and would result in a time complexity of $O(d)$, where d denotes the depth of the tree. A heuristic of having the longest node (containing maximum number of elements in a certain level) always first may increase the probability that a concept is always discovered in the left subtree.

To clarify the worst case, best case and the average case, let us assume that the partition tree has four levels ($d=4$) and the average branching factor $b=4$. The worst case traversal would result in exploring all $\frac{4^4 - 1}{4-1} = 85$ nodes where the best case is a traversal of only 4 nodes. For the average case, from each node, only the first half of the descendents is decided to be traversed. This would cause the `partitionTreeTraversal` algorithm traverse the nodes marked with thick black borders shown in Fig. 3. It should be noted that the average case is obtained only when the `searchVector` is organized in the first half of the descendents which results in a traversal of $\frac{2^{3+1} - 1}{2-1} = 15$ nodes of the partition tree (all these 15 nodes are marked with thick black borders in Fig. 3).

The function `partitionTreeTraversal` is written in such a manner that it avoids unnecessary branches while traversing the tree. For example, if the top level `searchVector` is *{French, Shushi, Barbeque}* then the traversal would follow the path depicted in Fig. 4 where the DFS traversal is marked by numbers 1 to 10. The partition tree is shaded in Fig. 4 and the search vector is drawn in black. From this, it becomes evident that the average case time complexity of the algorithm is highly dependent on the dispersed characteristic of the `searchVector`. At *level 1* the whole `searchvector` is found; the algorithm then searches the entire `searchVector` in the left-most branch of *level 1*, but it gets only a subset of `searchVector`, this traversal is propagated upto *level 4* to update the summarization of corresponding α-value for only relevant part of the original `searchVector`, i.e. for *{French}*.

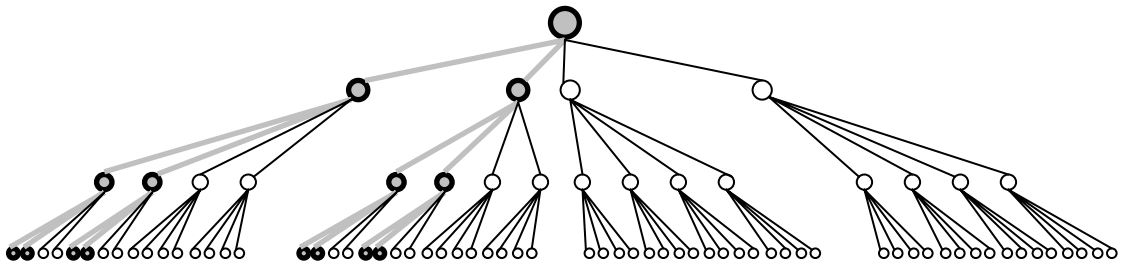

Fig. 3. Best case traversal with four levels and a branching factor of 4.

Level 3 contains pointers to children that are *{Φ}*, hence a traversal of *level 4* indicates search success of an atomic concept in *level 3*. The algorithm then searches for *{Shushi, Barbeque}* in the middle branch at *level 2*, where again a subset *{Shushi}* is detected. The search fails in *level 3*, in the left branch as it contains only *{Chinese}*, the algorithm immediately returns to *level 2* for the next branch and succeeds for *{Shushi}*

that goes upto *level 4*. The algorithm then continues the searching taking *{Barbeque}* alone, at the right most branch of *level 1* in the same way. It is evident from Fig. 4 that the unnecessary branches are always omitted by the `partitionTreeTraversal` algorithm. So the performance of the algorithm depends on the distribution of the `searchVector` in the partition tree. Average case is dominated by the probability of the existence of a `searchVector` in a certain node at certain level.

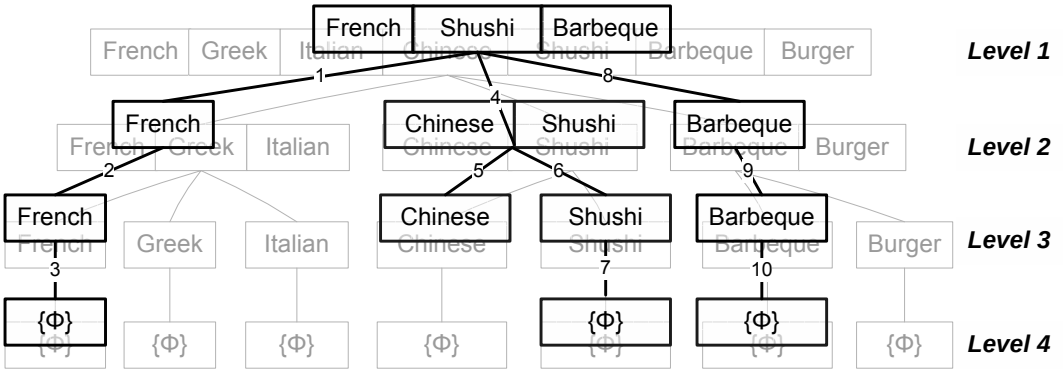

Fig. 4. Path traversed when the top level `searchVector` is *{French, Shushi, Barbeque}*. The partition tree is drawn in light gray. The DFS traversal is marked with numbers from 1 to 10.

Now consider Fig. 5 where the probability of availability of *{Italian, Chinese, Shushi, Barbeque}* in the root node is 1 as there is only a single node in that level. The probability that *{Barbeque, Chinese, Italian, Shushi}* is entirely included in one node of *level 2* is $\frac{1}{3} * \frac{1}{15} = \frac{1}{45}$ as there are three nodes in this level and the number of subsets in the power set of the `searchVector` is *15* excluding the empty set. The probability that any subset from this power set would avail itself entirely in one node of this level is $\frac{1}{45}$. Again let us consider *level 2* where *{Chinese, Shushi}* is found in the 2nd node of Fig. 5. Hence the probability that *{Chinese, Shushi}* would be found entirely in one of the descendents is $\frac{1}{2} * \frac{1}{3} = \frac{1}{6}$ because the probability that any of the three subsets of *{Chinese, Shushi}* would be found in one of the two descendents is obviously $\frac{1}{6}$. Each time the algorithm traverses to a deeper node it enhances the probability depending on the domain length of the power set and the number of descendents. Hence in a certain node the probability that a subset of the `searchVector` would be found in a descendent is $\frac{1}{b \times (|D| - 1)}$ where *D* is the power set of the `searchVector` and $|D|$ is considered to be the number of subsets in the power set of the `searchVector`. The factor $(|D| - 1)$ appears from the fact that we do not need to employ the subset search for the empty subset of the power set *D*, and *b* represents average branching factor.

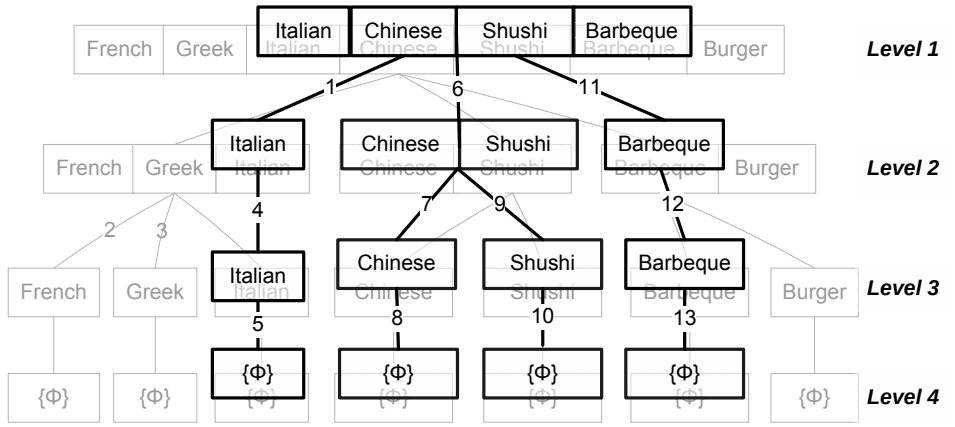

Fig. 5. `searchVector` *{Italian, Chinese, Shushi, Barbeque}* in the partition tree. The traversal is marked from 1 to 13. The solid black hierarchy indicates only the subsets that are searched upto the lowest level of the hierarchy.

3.2 Impact of the Character of the Similarity Tree on the Algorithm Performance

In our opinion, the performance of the `partitionTreeTraversal` algorithm is influenced by two factors: (1) degree of imprecision reflected by the number of entered attribute values (i.e. by their number, and their similarity), and (2) the character of the partition tree specified for this attribute. This section includes some tests on the performance of the proposed algorithm when applied with different types of similarity trees.

Data Set and Similarity Trees. The experiments have been conducted using some artificial datasets and different types of similarity trees. Data were picked randomly from the domain (domain size = 32 symbolic values), where the domain is defined as the set of all values specified at the lowest (i.e. $\alpha=1.0$) level of the partition tree. Number of values in a fuzzy record (stored during defuzzification in `searchVector`), considered as 75% imprecise, was kept at the 75% the domain size (i.e. 24 values) to allow for the randomness in choice of values, when maintaining consistent number of values. The number of records in each imprecise data file was 30000, so that the random characteristic could be evenly distributed among the partition tree as the `partitionTreeTraversal` algorithm omits the unnecessary branches. Hence the huge number of records ensures an average case traversal for the comparison purpose in this experiment. Algorithm run times for different percentage of imprecise records are recorded and plotted for different hierarchies.

The numeric values in the partition tree of Fig. 6(a) are symbols to represent non-numeric descriptive values like all other partition trees of the previous sections. Fig. 6(a) contains two hierarchies with different α-values (*H1* and *H2*) although their level-wise construction is common. *H1* is a distribution where the concepts are nested at the higher levels and *H2* is a distribution where concepts are nested at the lower levels of the hierarchy according to the α-values. The same domain is used for different types of hierarchies in Fig. 6 (b), (c), (d) and (e) although the numeric symbols are not explicitly presented in the trees. Fig. 6 portrays a total of eleven similarity hierarchies (*H1, H2, ..., H11*) which can be grouped in five categories and for each of the category only a single similarity tree is drawn. *H1* and *H2* have a total of five levels in their hierarchy but they differ in the α-values. Similarly, *H4*, *H5* and *H6* have three levels where *H7* to *H10* contain four levels in their hierarchies. The hierarchy *H3* of (b) has only two levels (the root and the leaves), whereas the hierarchy *H11* of (e) is a dendrogram with a total of 32 levels. Some levels in the hierarchy of Fig. 6 (e) have been omitted due to the limited space of this presentation. Obviously, Fig. 6 (e) presents the worst case possible among all the cases of Fig. 6 where every hierarchy contains 32 concepts in the lowest level of the hierarchy. The performance behavior is explained in the following subsection.

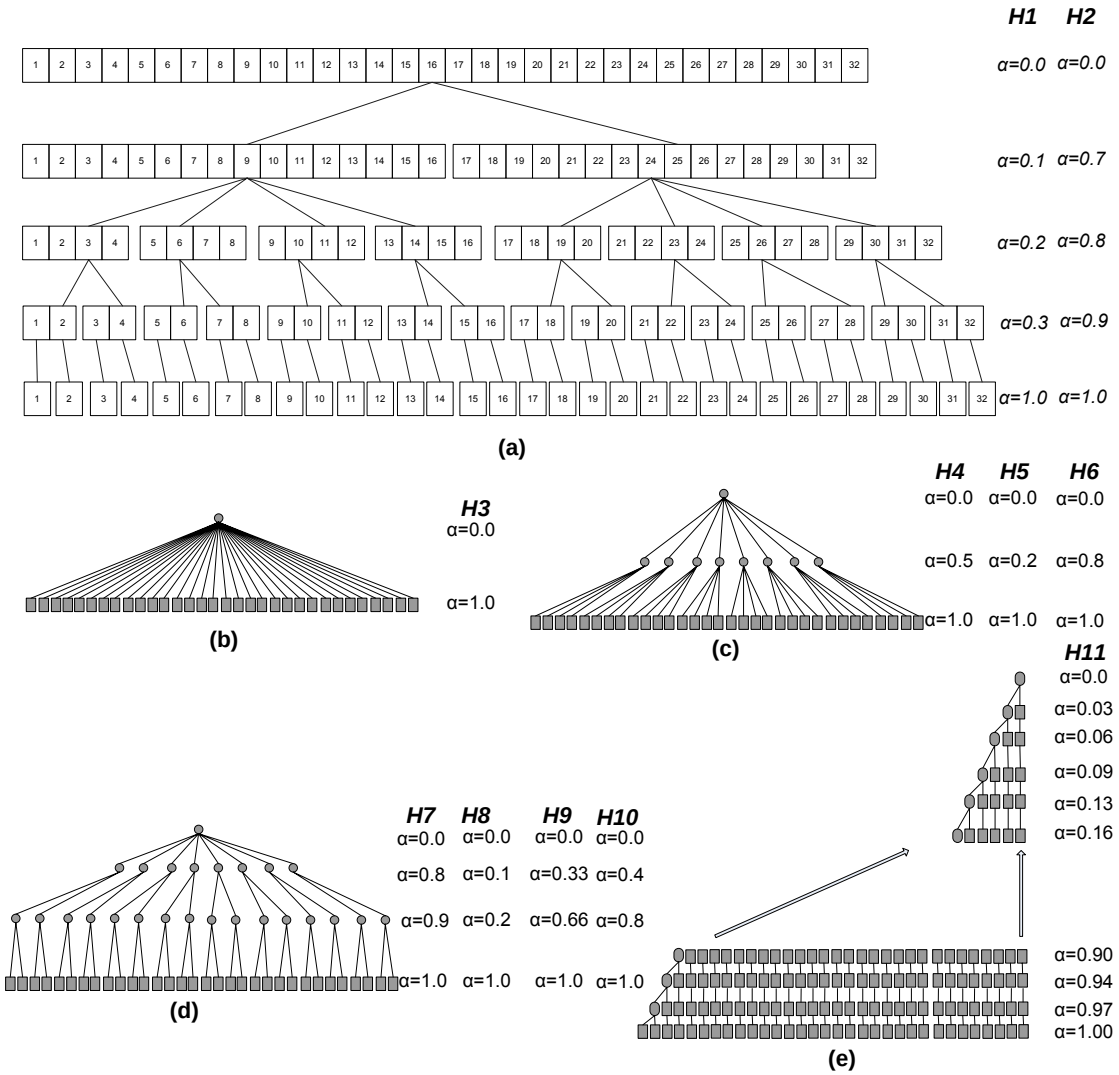

Fig. 6. Hierarchies used for the test. The hierarchies can be grouped as **(a)** *H1*, *H2*, **(b)** *H3,* **(c)** *H4*, *H5*, *H6*, **(d)** *H7, H8, H9, H10* and **(e)** *H11*. Each group possesses the look-alike partition tree with different α-values.

Behavior of `partitionTreeTraversal` Algorithm. The `partitionTreeTraversal` algorithm is applied to different types of hierarchies of Fig. 6 with different percentage of imprecision of data. The time required for the algorithm to traverse the partition tree is proportional to the percentage of imprecision of the data. Fig. 7 illustrates timing with two look-alike hierarchies, *H1* and *H2* with different nested characteristic depending on the α-values of Fig. 6(a). It is evident from Fig. 7 that the structure of these two hierarchies are almost the same because the `partitionTreeTraversal` algorithm traverses the same nodes while storing and updating reported existence of concepts in the hierarchy. This is despite the fact that the generated vote's fractions may be different, due to different propagation of α's within the tree. The algorithm is tested using nine other hierarchies depicted in Fig. 6. Fig. 8(a) is a plot of time at different percentage of imprecision of data with the hierarchies *H3* to *H10* of Fig. 6.

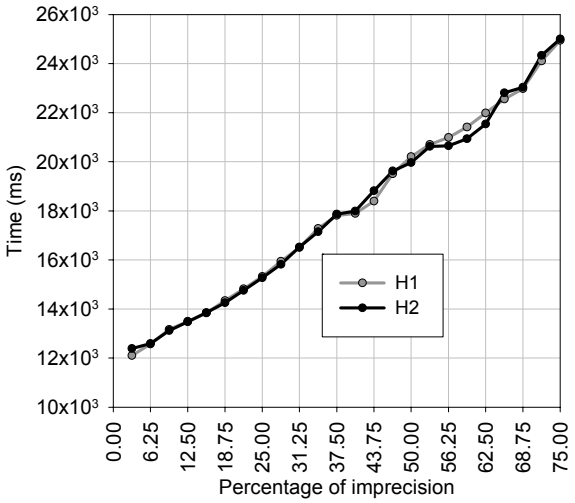

Fig. 7. Comparison of timing with two hierarchies that have the same distribution of concepts but different α-values. The hierarchies *H1* and *H2* are drawn in Fig. 6(a).

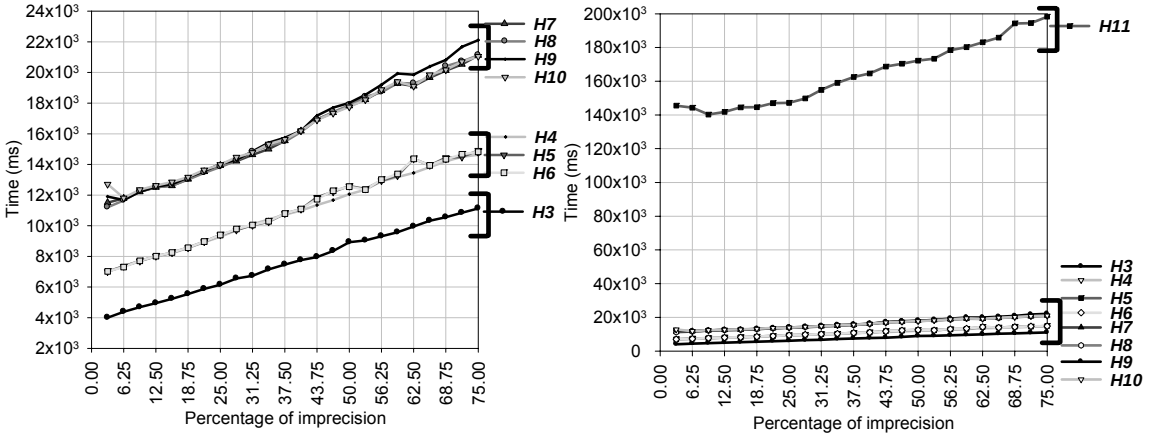

Fig. 8. (a) Plot for hierarchy *H3* to *H10* of Fig. 6, **(b)** Plot for hierarchy *H3* to *H11* of Fig. 6.

It is evident from Fig. 8(a) that the distribution of α-values among the levels does not have significant impact on the `partitionTreeTraversal` algorithm. Because the algorithm traverses the partition tree discretely depending on the number of levels rather than traversing in a continuous domain of α-values. A different algorithm that traverses depending on the continuous domain of α-values (rather than depending on the number of levels) better performs for the hierarchies that have nesting at the top levels for a downward search in the partition tree as it discovers most of the levels at the beginning of the domain of α-values. The timing plots of *H4*, *H5* and *H6* follow almost the same trend in the graph. Similarly, *H7*, *H8*, *H9* and *H10* also produce similar graphs that are closely aligned in Fig. 8(a). The trend of *H3* is separate from the other lines as it is a different hierarchy and performs the best as it has least number of levels in its similarity tree. As `partitionTreeTraversal` algorithm is a level dependent depth-first search, it performs the same both for the hierarchies nested at high levels and hierarchies nested at low levels. This behavior is reflected in Fig. 8(a) because the hierarchies with the same tree are clearly grouped in the plot. The plot has three groups because it includes three types of partition trees, (b), (c) and (d) of Fig. 6. The three types of hierarchies are well separated in the plot of Fig. 8(a).

Fig. 8(b) shows that the hierarchy *H11* takes the worst possible time compared to all other hierarchies (*H3* to *H10*). From the experiment, it becomes apparent that `partitionTreeTraversal` is a number of levels-dependent search, where the performance decreases with increase of the number of levels in the partition tree which suggests flattering the partition trees (by making them more bushy), or inclusion of new concepts without increment in the number of levels.

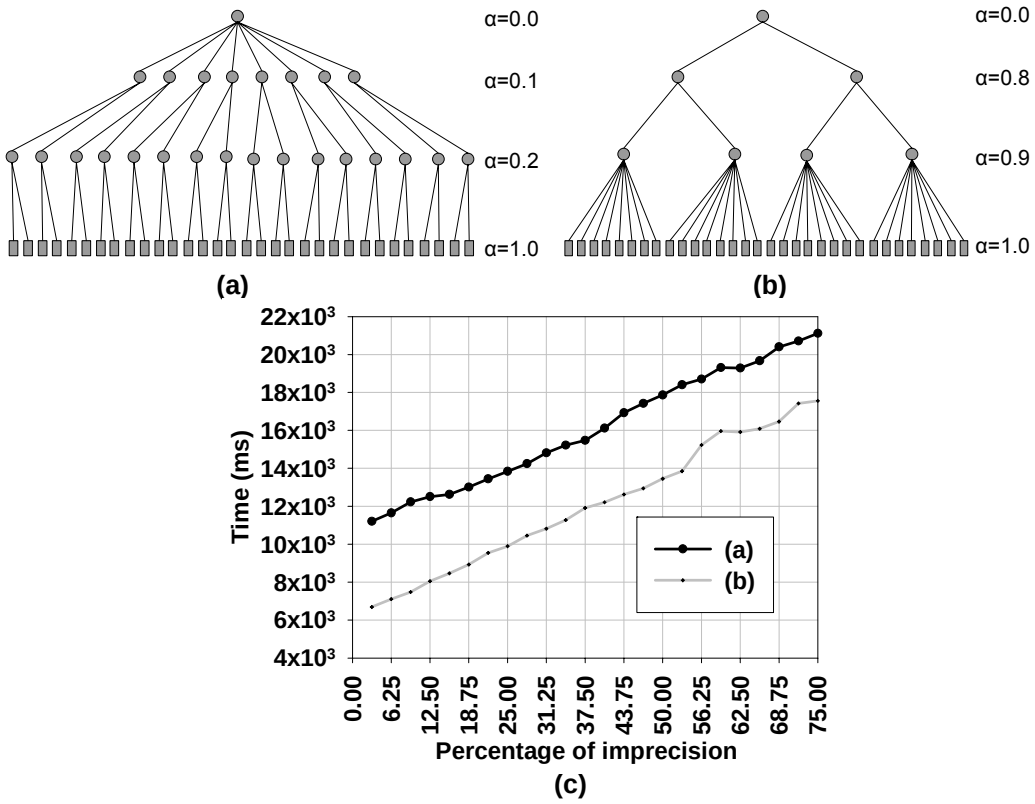

Fig. 9. (a) A partition tree where the abstracts start to split at high levels. **(b)** A partition tree where the abstracts split at low level. **(c)** Plot of runtime behaviors of `partitionTreeTraversal` algorithm with the hierarchies of (a) and (b) at different percentage of imprecision of data.

Impact of Conceptualization at Higher Levels and Lower Levels. Let us consider the partition trees of Fig. 9(a) and (b). The hierarchy of (a) starts splitting at high levels of the tree generating low α-values whereas (b) splits at the comparatively low level. Although both the hierarchies have 32 leaf nodes at the bottom of the tree, in (b), they are concentrated fast at the bottom of the tree. In contrary, the other hierarchy possesses more concepts relatively at the top of the tree resulting in more intermediate nodes at every level. The `partitionTreeTraversal` algorithm would take more time in the case of Fig. 9(a) compared to (b) as the algorithm needs to traverse more nodes for traversing the hierarchy of (a). The resulting runtime behaviors with these two hierarchies with respect to percentage of imprecision of data are plotted in Fig. 9(c).

Fig. 9(c) shows that the `partitionTreeTraversal` algorithm performs better with the hierarchy of Fig. 9(b) than that of (a). This suggests low branching factor near the root of the partition tree and comparatively higher branching factor at the bottom.

4. Conclusions

The work presented in this paper shows that the fuzzy collections can be transferred to the atomic values in the efficient way. The heuristic algorithm presented here allows for transfer of fuzzy records that reflect uncertainty to the form that allows analysis of such data via majority of regular data mining algorithms. Although, the presented heuristic for counting fractions of records as frequencies of observed atomic values might be questioned by statisticians, the technique in our opinion can be treated as a "proof of concept" that imprecise data does not has to be disregarded, but can be mined in a regular matter. Task of discovery of the most appropriate data defuzzification algorithms remains as the topic for future research.

References

[1] J.C. Cubero, N. Marín, J..M. Medina, O. Pons, M.A. Vila, "Fuzzy object Management in an Object-Relational Framework", X Intl. Conf. of information processing and management of uncertainty in knowledge-based systems, pp.1767--1774. 2004.

[2] Z. Ma (Ed.), Advances In Fuzzy Object-oriented Databases: Modeling And Applications, Idea Group Publishing, Hershey, PA, USA, 2004.

[3] F. Berzal, N. Marín, O. Pons, M.A. Vila, "Development of applications with fuzzy objects in modern programming platforms", International Journal of Intelligent Systems, Vol. 20, Issue 11, 09/2005, pp. 1117 – 1136.

[4] B.P. Buckles & F.E. Petry, "A fuzzy representation of data for relational databases", Fuzzy Sets and Systems, 7(3), 1982, pp. 213-226.

[5] F.E. Petry, *Fuzzy Databases: Principles and Applications*, Kluwer Academic Publishers, Boston, MA, 1996.

[6] S. Shenoi and A. Melton, "Proximity Relations in the Fuzzy Relational Database Model", *International Journal of Fuzzy Sets and Systems, 31*(3), 1989, pp. 285-296.

[7] S. Shenoi, A. Melton, and L. T. Fan, "Functional Dependencies and Normal Forms in the Fuzzy Relational Database Model", *Information Sciences, 60*(1-2), 1992, pp. 1-28.

[8] R. Angryk, "Similarity-driven Defuzzification of Fuzzy Tuples for Entropy-based Data Classification Purposes", Proceedings of the 15th IEEE International Conference on Fuzzy Systems (FUZZ-IEEE '06), a part of 2006 (4th) IEEE World Congress on Computational Intelligence, Vancouver, Canada, July 2006, pp. 1490-1498.

[9] R. Angryk, "On Interpretation of Non-Atomic Values and Induction of Decision Rules in Fuzzy Relational Databases", *Proceedings of the 8th International Conference on Artificial Intelligence and Soft Computing (ICAISC '06)*, Zakopane, Poland, June 2006, published by L. Rutkowski, R. Tadeusiewicz, L.A. Zadeh, J. Zurada (Eds.) in Series: *Lecture Notes in Artificial Intelligence (LNAI)*, *Vol. 4029*, Springer-Verlag, 2006, XXI, 1235 p., ISBN: 3-540-35748-3, pp. 170-181.

[10] R. Angryk, F. Petry, R. Ladner, "Mining Generalized Knowledge from Imperfect Data," *Proceedings of the 10th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU '04)*, Perugia, Italy, July 2004, pp. 739-746.

[11] R. Angryk, F. Petry, "Discovery of Abstract Knowledge from Non-Atomic Attribute Values in Fuzzy Relational Databases," in: B. Bouchon-Meunier, G. Coletti, R. Yager (Eds.), *Modern Information Processing, From Theory to Applications*, Elsevier, 2006, X, 480 p., Hardcover, ISBN: 0-444-52075-9, pp. 171-182.

[12] M. Kantardzic, J. Zurada, New Generation of Data Mining Applications, IEEE Press and John Wiley, February 2005, 696 p., ISBN: 0-471-65605-4

[13] L.A. Zadeh, "Similarity relations and fuzzy orderings", *Information Sciences, 3*(2), 1970, pp. 177-200.

[14] H.– H. Bock, E. Diday (Eds.), "Analysis of Symbolic Data, Exploratory Methods for Extracting Statistical Information from Complex Data", Springer 2000, XVIII, 425 p., ISBN: 978-3-540-66619-6

[15] Bertrand, P. & Goupil, F. (2000), 'Descriptive statistics for symbolic data', *Analysis of Symbolic Data: Exploratory Methods for Extracting Statistical Information from Complex Data* (eds. H.-H. Bock and E. Diday), Berlin, Springer-Verlag, pp 103-124.