2019 IEEE Symposium Series on Computational Intelligence (SSCI)
December 6-9 2019, Xiamen, China

Associative Data Model in Search for Nearest Neighbors and Similar Patterns

Adrian Horzyk
AGH University of Science and Technology in Krakow
Depart. of Biocybernetics and Biomedical Engineering
30-059 Kraków, Poland
Email: horzyk@agh.edu.pl

Janusz A. Starzyk
University of Information Technology and Management
in Rzeszow, Department of Applied Information Systems
35-225 Rzeszów, Poland
Email: starzykj@gmail.com

***Abstract*—This paper introduces a biologically inspired associative data model and structure for finding nearest neighbors and similar patterns. The method can be used as an alternative to the classical approaches to accelerate the search for such patterns using various priorities for attributes according to the Sebestyen measure. The presented structure, together with algorithms developed in this paper can be useful in various computational intelligence tasks like pattern matching, recognition, clustering, classification, multi-criterion search etc. This approach is particularly useful for the on-line operation of associative neural network graphs. Graphs that dynamically develop their structure during learning on training data. The results of experiments show that the associative approach can substantially accelerate the nearest neighbor search and that associative structures can also be used as a model for KNN tasks. Finally, this paper presents how the associative structures can be used to self-organize data and represent knowledge about them in the associative way, which yields new search approaches described in this paper.**

Keywords—nearest neighbor search; similarity; classification; associative data model; associative data structures.

I. Introduction

The most common way of storing data is the use of tables, where various objects are represented by records of values of various attributes. Such objects can be sorted by only one chosen attribute. So when we need to find similar objects or determine its nearest neighbors, the whole table must be searched, and the distances to specific data must be computed and compared for all objects. This approach is not efficient and is the bottleneck of the k nearest neighbors (KNN) classifiers [1-3]. The computational complexity of the search increases at least linearly with the number of compared objects in the table storage.

Our brains also have the ability to find similar objects, but they undoubtedly do not search through long lists of known objects because our slowly working neurons (spiking maximally hundred times per second [4] which is very slow in comparison to CPU clocks of contemporary computers) will not be able to find similar objects in an acceptable time. This suggests that biological neural structures of our brains use a different approach to finding similar objects. Taking into account the slow spiking speed of biological neurons, the biological neuronal structures must organize representation of similar objects close to each other to be able to find them quickly.

The most important goal of this paper is to present the associative model and structures for storing similar objects (patterns) closely and making them quickly available. The efficiency of this model is compared to the classic approaches where tables are used [5]. We present three algorithms using associative structures that search for similarities of objects and nearest neighbors faster than the classic KNN algorithm. These algorithms also improve the associative approach presented in [6], where the first attribute was always used for the searches considered in this paper. Moreover, we show how we can change the results of such searches using various priorities for attributes, and how this improves the classification results of k nearest neighbor classifiers.

II. Associative Data Model and Structures

The Associative Graph Data Structure (AGDS) was inspired by the analysis of connections between real neurons and developed by Horzyk [5], [7]. This graph structure consists of vertices representing unique attribute values, objects defined by these attribute values, connections that link the nearest values of each attribute, and links that connect the object to attribute values that define them. The values are represented by **value vertices**, and objects are represented by **object vertices**. Duplicates of identical values of each attribute are aggregated and represented by the same value vertices. Identical objects that appear in the input data are also represented by the same object vertices (Fig. 1). Such aggregations of various duplicates reduce the number of value vertices and object vertices appearing in tables.

Thanks to the use of graph structure, instead of a tabular one, all values of each attribute can be sorted simultaneously, which accelerates various search operations processed on AGDS structures. The sorting and aggregation of attribute values are beneficial. For instance, it is possible to move fast between objects defined by the same or close values.

For example, consider Associative Graph Data Structure created on 30 samples from Fisher's Iris data set presented in Fig. 1. In this figure, objects O21 and O24 are connected to the same attribute value 1.7 for the petal length and to close value vertices 5.1 and 5.4 for the sepal length or 0.2 and 0.5 for the petal width, which reduces the graph paths between similar objects and allows various search algorithms to find similar objects faster

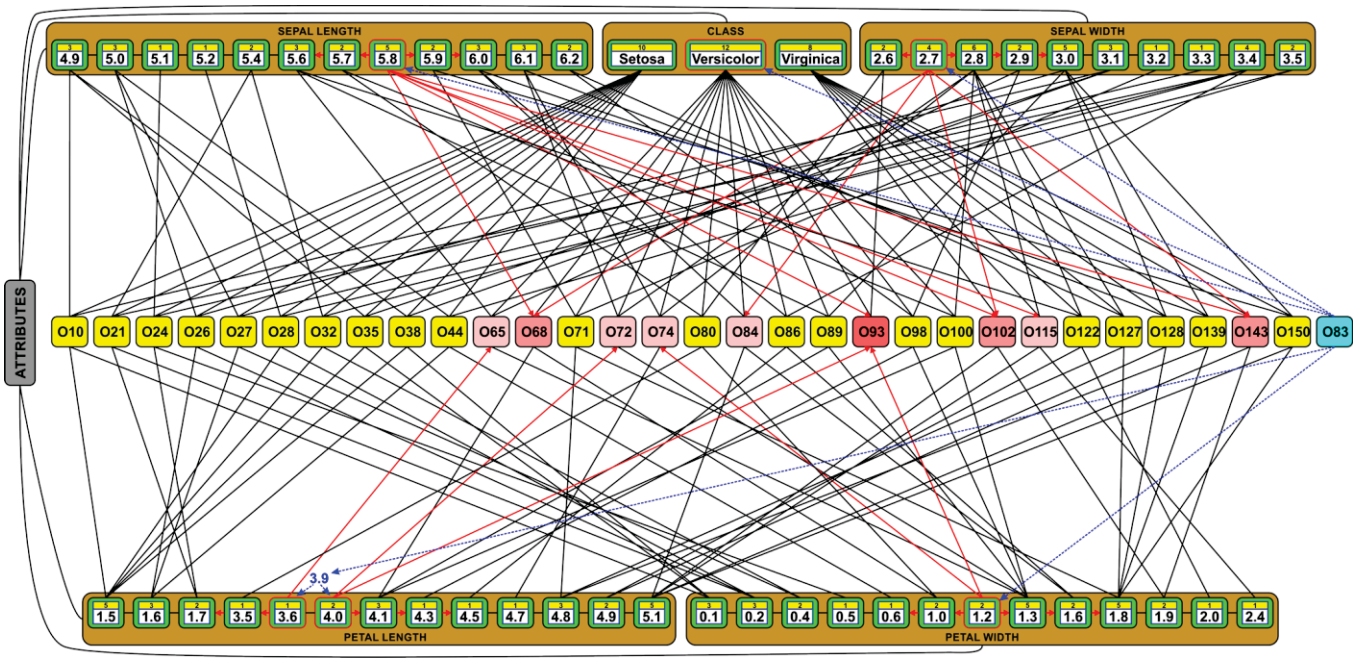

Fig. 1 Associative Graph Data Structure created on sample data (30 selected Iris samples) with visible aggregations of representation of the same attribute values and counted duplicates represented by value vertices. They are connected to their neighbors in the sorted order and used to define various objects represented by the object vertices. This structure is used for searching for k nearest neighbors for the object O93 starting from the closest attribute values [5.8, 2.7, 3.9, 1.2] that define this object and spreading out to other connected close values and to the objects (training samples) that constitute this structure.

than using classic approaches. These properties of the proposed approach will be used in the following subsections to define faster associative algorithms for searching for k nearest neighbors.

The AGDS structure is not only a different representation of the dataset stored in the table, but this graph structure represents much more useful relations between data than the tabular structure. We can quickly move between similar attribute values and find all objects defined by them as well as moving from one object to another using various criteria. The AGDS structure represents not only definitions of objects like tabular structures do, but also various relations between these objects. Many algorithms search through data tables, looping those tables many times to find necessary relationships. When using associative structures like AGDS, we do not need to loop because some relations are immediately available through AGDS associations. This paper uncovers only a small part of the abilities of these graph structures, focusing on the efficient search algorithms for similar (nearest neighbors) objects.

III. Nearest Neighbors Search

The search of nearest neighbors is one of the approaches used for object matching, recognition, clustering, or classification because nearest neighbors, i.e. the most similar objects in a given distance measure, intuitively suggest the grouping and labeling conditions for the objects. This approach is used in the different variations of the k nearest neighbors' methods [2], [8-10], so they are used in this paper for testing and comparisons of the introduced approach. The following subsections will discuss and compare the classic approach based on the tabular data representation to the proposed associative approach using the AGDS structure.

The main idea of the associative approach is to limit the number of objects (training samples) that must be selected to compute distances to them in order to find k nearest neighbors. It is possible thanks to the associative structure where similar object vertices are connected through sorted and aggregated value vertices. If fact, the search for k nearest neighbors proceeds only in the closest surrounding of the classified sample leaving out all distant objects.

A. Classic KNN Using a Data Table and Rank List of Objects

The classic KNN approaches [1-3], [8-9] use a data table where all objects (training samples) are stored. For a given input object, for which we would like to find k nearest neighbors (k most similar objects), the algorithm loops through the whole collection computing the chosen distance (e.g. Euclidean, Manhattan, Minkowski, Sebestyen, or Mahalanobis distance) between the input object and all objects stored in the table [11]. Next, it compares the computed distances with the distances of the currently closest objects stored in the rank list and inserts those which are closer than the most distant object in the rank list. The k closest objects in the rank list are updated during the loop over all objects. First, a new computed distance between the input object and the next checked object in the tabular collection is compared to the most distant object in the rank list. When this distance is smaller than the one of the most distant object, then the algorithm loops through the rank list from its end to its beginning and finds the right place to put there this closer object. It usually overfills the rank list (over k elements), so the last, most distant object is removed from the rank list if only it is more distant than the k-th element in the rank list. If it is not, all objects with the same distance are stored in the overlengthen rank list until the k-th element is smaller than the last element in this list. When it finally happens, then all objects

with the same distance which overlengthen the list are removed at the same time. The algorithm is running until all objects (training samples) compute their distance and check them with those in the rank list. The final rank list points out the k or sometimes more (if there are a few objects with the same distance at the end of the rank list, so we cannot neglect them) closest objects (nearest neighbors) in the collection.

These objects are finally used to classify the input sample based on the most frequent class represented by the object in the rank list. If there are no winning class, or when the most frequent class represented by the objects in the rank list is different from the class defined for the input object, the classification is incorrect.

Another classic approach to KNN classification requires to compute distances of the input object to all training objects (samples) in the collection, and then sort these distances in the ascendant order, e.g. using quicksort. Next, the k-th or more (if there are a few objects with the same distance at the end of the rank list) closest objects are taken as the k nearest neighbors and determine the final classification. This approach is slower than the previously described one using rank list because it requires to sort all objects, while we directly use only k-th closest ones.

The bottleneck of the classic approaches using a table for storing training samples is that all samples must always be looped, and to all of them, the distance must be computed, compared, and sorted to find the k nearest neighbors. The associative approach presented in the following subsections shows how the training objects (samples) can be associatively organized to avoid looping through all of them to find the k nearest neighbors (the k most similar objects) in the training dataset.

Algorithm 1:

```
FindKNN(k, dataRowNew) // find k nearest neighbors
oRank = new List<CDRow>() // Create rank list (Part1)
dMaxoRank = 0.0 // max distance in oRank
countoRank = 0 // count of the oRank elements
foreach (dataRow in dataTable) // loop over all rows
d = EuclideanNormDist(dataRow, dataRowNew) // Part2
if ((d <= dMaxoRank) || (countoRank < k)) // Part3
  i = countoRank-1
  if (i >= 0)
  do
    if (oRank[i].d > d) i--
    else break
  while (i >= 0)
  oRank.Insert(i+1, new CDRow(d, dataRow))
  countoRank++
  if (countoRank > k)
  if (oRank[k-1].d < oRank[k].d) // Part4
    oRank.RemoveRange(k, countoRank-k)
    countoRank = k
  dMaxoRank = oRank.Last().d
return GetWinningClass(oRank)

EuclideanNormDist(obj1, obj2)
dist = 0.0;
for (int i = 1; i < obj1.NoAttributes; i++)
  d += Pow(obj1[i] - obj2[i]) / attrRanges[i], 2);
return Sqrt(d);
```

where the function GetWinningClass points out the class that is most frequently represented by the objects in the rank list oRank or returns null if there is no winning class.

In this algorithm, the rank list oRank is created (Part1) to store the sorted list of k nearest objects accordingly to the Euclidean normalized distance computed by the EuclideanNormDist function (Part2), where the distances are computed on the basis of the distances normalized by the ranges of attributes. If the computed distance to the next object (dataRowNew) is less than the most distant object in the rank list oRank and the list already contains k elements (Part3), then it is inserted in the right order into this list. Next, when the rank list becomes longer than k and the k-th object has a different distance than the last objects (Part4), the most distant objects are removed from the rank list. Finally, after looking through all objects, we get the final list of k nearest objects (neighbors) or more if the k-th object in this list has the same distance to the source object as the subsequent objects in this rank list.

B. Associative KNN based on the Most Variant Feature

The first associative approach (AKNN-1) uses only one of the most variant attributes to search for k nearest neighbors. The most variant attribute has the biggest number of unique values, i.e. the least number of duplicates in the tabular collection (e.g. both attributes petal length and petal width have 13 unique values in Fig. 1, and that is more than the other attributes have).

We select one of the most variant attributes because their value vertices are connected on average to the smaller number of object vertices than the value vertices of the attributes with less number of represented unique values in the vertices. Thanks to it, we can substantially reduce the number of visited object vertices for which the distances to the classified object (e.g. object O93 in Fig. 1) are computed. The number of unique values for each attribute is counted during the construction of the AGDS structure for a given dataset, so we need only to find the most numerous ones. This can be done in constant time to start the search.

The associative search process starts from the value vertex of the chosen most numerable attribute that defines the classified object. In the sample of thirty Iris objects (Fig. 1), we can choose petal length or petal width as this most numerable attribute. According to which one will be chosen, the search process will start from value vertex 4.0 for petal length or from value vertex 1.2 for petal width because these values are the closest to those defining the classified object O93. Next, all connected object vertices to the chosen nearest value vertex are visited, and their distances to the classified object are computed. These distances are compared to the most distant object in the rank list (if there are any) and those which are closer than the most distant one (or the rank list has no k objects yet) are put into the rank list in the ascendant order, starting the search for this position from the end of this rank list.

Next, the algorithm moves from the current value vertex to the next smaller or greater value vertex that has not been visited yet and represents the next closest value to the value of the classified object (i.e. 4.1 for the petal length or 1.3 for the petal width in Fig. 1). For this next value vertex, the whole process (that computes distances of objects represented by the object vertices connected to this value vertex and updating the rank list) is repeated until the distance of the value represented by the next value vertex to the value of the classified object is greater than the distance of the most distant object in the rank list. This is the stop condition of this algorithm because if this condition is true, we can be sure that there cannot be another object that will be

closer to the classified one. When the stop condition is satisfied, the rank list contains k nearest neighbors with their computed distances, and they can be used to compute the final result of the classification of the KNN algorithm, or these neighbors can be used to other issues. Thus, the result of the associative search for nearest neighbor is the same as for the classic KNN algorithm, but the nearest neighbors are usually found much faster than using the classical KNN approach and a table for storing objects because we do not loop through all objects.

Algorithm 2:

```
FindAKNN(k, dataRowNew) // Associative KNN using one attr
range = attrRanges[noMNA] // most numerous attr range
closestVV = AVBT[noMNA].FindClosestVV(dataRowNew[noMNA])
rowVVVal = dataRowNew[noMNA]
if (closestVV.Val <= rowVVVal) // Part1
  smallerVV = closestVV
  dToSmallerVV = (rowVVVal - smallerVV.Val) / range
  greaterVV = smallerVV.Next
else
  smallerVV = null
  dToSmallerVV = Double.MaxVal
  greaterVV = closestVV
if (greaterVV != null)
  dToGreaterVV = (greaterVV.Val - rowVVVal) / range
else dToGreaterVV = Double.MaxVal
ovRank = new List< COVRankEl>() // Part2
ovRankCount = 0
maxovRank = 0.0
do
  foreach (ov in closestVV.OVs) // Part3
    d = EuclideanNormDist(ov, dataRowNew)
    if ((d <= maxovRank) || (ovRankCount < k))
      i = ovRankCount - 1
      FindovRankPos(d, i)
      ovRank.Insert(i+1, new COVRankEl(d, ov))
      ovRankCount++
      if (ovRankCount > k)
        if (ovRank[k-1].d < ovRank[k].d)
        ovRank.RemoveRange(k, ovRankCount - k)
        ovRankCount = k
      maxovRank = ovRank.Last().d
  if (smallerVV == closestVV) // Part4
    smallerVV = smallerVV.Previous
    if (smallerVV != null)
      dToSmallerVV = (rowVVVal - smallerVV.Val) / range
    else
      dToSmallerVV = Double.MaxVal
      if (greaterVV == null) break
  else
    greaterVV = greaterVV.Next
    if (greaterVV != null)
      dToGreaterVV = (greaterVV.Val - rowVVVal) / range
    else
      dToGreaterVV = Double.MaxVal
      if (smallerVV == null) break
  if (dToSmallerVV <= dToGreaterVV)
    closestVV = smallerVV
  else closestVV = greaterVV
while ((dToSmallerVV <= maxovRank ||
       dToGreaterVV <= maxovRank) || (ovRank.Count < k))
return GetWinningClass(ovRank)

FindovRankPos(d, i)
if (i >= 0)
  do
    if (ovRank[i].d > d) i--
    else break
  while (i >= 0)
return i
```

In Algorithm 2, we start from the closest value vertices (closestVV) to the source object represented in this associative structure supported by AVB+trees (AVBT) [5] used to efficiently store and search for value vertices (VV). Next, two neighbors (a smaller or equal one (smallerVV) and a greater one (greaterVV)) are established (Part1) together with the distances (dToSmallerVV and dToGreaterVV) to these neighbors. Afterward, the rank list ovRank is created (Part2) and is filled with the nearest objects represented in the associative structure.

The nearest objects are established on the basis of checking only the objects connected to the nearest values of the selected attribute to the value of this attribute representing the source object (dataRowNew). Thus, we check all objects connected to the given nearest value of the selected attribute (Part3), and next move to the next closest value (smaller or greater one) (Part 4), and calculate Euclidean normalized distances (EuclideanNormDist) to the source object (dataRowNew). We continue this process until the next closest value is more distant to the value of this attribute of the source object than the most distant (k-th) object in the rank list because the further search will not bring closer objects and do not change the rank list anymore.

C. Associative KNN with a Rank List of Closest Features

The second associative approach (AKNN-L) uses the same AGDS structure as before, but it uses several most numerous attributes (e.g. these attributes which numbers of unique values are greater than the average number of unique values for all attributes). In this approach, we use another rank list of the closest values of those most numerous attributes to the values of the classified object.

During the main loop of this algorithm, the next closest value vertex is chosen from this rank list until the same stop condition as before is achieved, but the distances of the objects represented by the visited object vertices to the classified object are computed gradually, not at once. Namely, suppose that we have J most numerous attributes. When the object vertex is visited for the j time (where j < J) for the i-th attribute, its distance to the classified object is only partially computed for the attribute value that pointed this object vertex, e.g. (1) for Euclidean distance, (2) for Euclidean distance, (3) for Minkowski distance, and (4) Sebestyen distance:

$$d \mathrel{+}= |x_i - \hat{x}_i| \tag{1}$$

$$d \mathrel{+}= (x_i - \hat{x}_i)^2 \tag{2}$$

$$d \mathrel{+}= (x_i - \hat{x}_i)^m \tag{3}$$

$$d \mathrel{+}= (x_i - \hat{x}_i) w_i (x_i - \hat{x}_i)^T \tag{4}$$

where w_i is a diagonal matrix of weights.

Hence, we do not lose time for computing the whole distance for the visited object vertex because it might not be one of the nearest neighbors. Only when the object vertex is visited the J time for the i-th attribute, the distance is finally computed, e.g. (5) for Euclidean distance, (6) for Euclidean distance, (7) for Minkowski distance, and (8) Sebestyen distance

$$d = d + \sum_{k \in \{i, A^U\}} |x_k - \hat{x}_k| \tag{5}$$

$$d = \sqrt{d + \sum_{k \in \{i, A^U\}} (x_k - \hat{x}_k)^2} \tag{6}$$

$$d = \sqrt[m]{d + \sum_{k\in\{i,A^U\}}(x_k - \hat{x}_k)^m} \quad (7)$$

$$d = \sqrt{d + \sum_{k\in\{i,A^U\}}(x_k - \hat{x}_k)w_k(x_k - \hat{x}_k)^T} \quad (8)$$

and we have compared with the distances in the rank list of nearest neighbors if only the computed distance is smaller than the distance to the most distant object in the rank list. If we proceed with this algorithm always for the closest value vertex for the selected most numerous attributes, the above-defined stop condition is achieved faster because we move along a few most numerous attributes, and the greatest distance in the rank list decreases faster. We additionally save some computational time for calculating only partial distances (1) – (4) of the objects that have only some attribute values close to the classified objects, but they are not the nearest neighbors because of the other attribute values that are distant.

On the other hand, we have to take into consideration the extra computational cost for handling the rank list of the closest values of the chosen subset of the most numerous attributes. That is why the following algorithm (Algorithm 3) uses several attributes defined by the most numerous number of unique values instead of only one attribute of the most numerous number of unique values used in Algorithm 2.

Algorithm 3:

```
FindAKNN-L(k, dataRowNew) // Assoc. KNN using several attr
ovRankCount = 0
vvRank = new List<VVRankEl>()
foreach (attrNum in descendantNoAttrValsList)
  noAttr = attrNum.noAttr
  closestVV = AVBT[noAttr].
              FindClosestVV(dataRowNew[noAttr])
  rowVVVal = dataRowNew[noAttr]
  if (closestVV.Val <= rowVVVal)  // Part1
    smallerVV = closestVV
    d = (rowVVVal - smallerVV.Val) / attrRanges[noAttr]
    i = vvRank.Count - 1
    FindvvRankPos(d, i)
    vvRank.Insert(i+1, new VVRankEl(d, smallerVV))
    greaterVV = smallerVV.Next
    if (greaterVV != null)
      d = (greaterVV.Val - rowVVVal) / attrRanges[noAttr]
      i = vvRank.Count - 1
      FindvvRankPos(d, i)
      vvRank.Insert(i+1, new VVRankEl (d, greaterVV))
  else
    greaterVV = smallerVV.Next
    if (greaterVV != null)
      d = (greaterVV.Val - rowVVVal) / attrRanges[noAttr]
      i = vvRank.Count - 1
      FindvvRankPos(d, i)
      vvRank.Insert(i+1, new VVRankEl(d, greaterVV))
closestVV = vvRank.First().vv
noCloseAttr = closestVV.AttrNo
rowVVVal = dataRowNew[noCloseAttr]
ovRank = new List<OVRankEl>()
maxovRank = 0
do       // Part2
  foreach (ov in closestVV.OVs)
    d = ov.AddDist((ov.VVList[noCloseAttr].Val - rowVVVal)
        / attrRanges[noCloseAttr], dataRowNew, attrRanges)
    if ((d >= 0)&& ((d <= maxovRank) || (ovRankCount < k)))
      i = ovRankCount - 1
      FindovRankPos(d, i)
      ovRank.Insert(i+1, new OVRankEl(d, ov))
      ovRankCount++
      if (ovRankCount > k)
        if (ovRank[k-1].d < ovRank[k].d)
          ovRank.RemoveRange(k, ovRankCount - k)
          ovRankCount = k
      maxovRank = ovRank.Last().d
  vvRank.RemoveAt(0)
  if (closestVV.Val <= rowVVVal)
    vv = closestVV.Previous
    if (vv != null)
      d = (rowVVVal - vv.Val) / attrRanges[noCloseAttr]
  else
    vv = closestVV.Next
    if (vv != null)
      d = (vv.Val - rowVVVal) / attrRanges[noCloseAttr]
  i = vvRank.Count - 1
  if (vv != null) //&& (d < maxovRank))
    FindvvRankPos(d)
    vvRank.Insert(i+1, new VVRankEl(d, vv))
  if (vvRank.Count == 0) break
  closestVV = vvRank.First().vv
  noCloseAttr = closestVV.AttrNo
  rowVVVal = dataRowNew[noCloseAttr]
while ((vvRank.First().d <=maxovRank) || (ovRank.Count <k))
return GetWinningClass(ovRank)

FindvvRankPos(d)
if (i >= 0)
  do
    if (vvRank[i].d > d) i--
    else break
  while (i >= 0)
return i

AddDist(distVV, newDataRow, attrRanges)
if (noOperation < noAGDSOperation) // Part3
  distOV = distVV * distVV
  cntrAdd = 1
  noOperation = noAGDSOperation
else // Part4
  distOV += distVV * distVV
  cntrAdd++
  if (cntrAdd == noNumerousAttributes)
    for (vNo = 1; vNo < vvList.Count; vNo++)
      if (notNumerousAttributes[vNo])
        dist = (vvList[vNo].Value - newDataRow[vNo]) /
               attrRanges[vNo]
        distOV += dist * dist
    return Math.Sqrt(distOV)
return -1
```

Algorithm 3 works similarly to Algorithm 2, but we take into account several most invariant (having the greatest number of unique values) attributes. For such attributes, we search for the smaller and greater closest values to the values defining the source object (dataRowNew) (Part1), so the search is more efficient, and the main loop (Part2) finishes after a smaller number of steps than in Algorithm 2. However, there is an extra cost of updating the rank list (vvRank) of the closest values for the selected subset of the most invariant attributes. In this algorithm, Euclidean normalized distances are computed gradually (AddDist), adding the square of the distance for the currently considered attribute into the sum of squares (distOV) (Part3). This way we do not lose time for full computation of the distances for objects defined by the values of attributes which are distant from the values of the selected most invariant attribute values of the source object. Only for those objects which distances of their defining values of all selected most invariant attributes are close enough (Part4), the Euclidean normalized distances are finally computed, compared, and inserted into the rank list of the k-th nearest objects (ovRank). Thanks this mechanism of gradual computation of distances of objects, the algorithm proceeds a less number of operations on the rank list of objects (ovRank).

D. Associative KNN with a Rank Table of Closest Features

The third associative approach (AKNN-T) (Algorithm 4) uses the same subset of the most numerous attributes as the

previous algorithm (Algorithm 3) and works in the same way, but it uses a rank table instead of the rank list. The change is in the efficiency of the way how the rank table works in comparison to the rank list.

Both Rank List and Rank Table have their pros and cons and dependently on the training dataset either one or the second might be more efficient, however in the most cases (as will be shown in this paper), the rank table is computationally more efficient than the rank list.

Algorithm 4:

```
FindAKNN-T(k, dataRowNew) // Assoc. KNN using several attr
cntrAGDSOperation++
int ovRankCount = 0
vvSmallerDist = new double[noAttrs]
vvGreaterDist = new double[noAttrs]
for (i = 1; i < noAttrs; i++)
  rowVVVals[i] = dataRowNew[i]
  vvSmaller[i] = null
  vvSmallerDist[i] = 0
  vvGreater[i] = null
  vvGreaterDist[i] = 0
foreach (attrNum in NumerousAttrRank)
  noAttr = attrNum.noAttr
  closestVV = AVBT[noAttr].
              FindClosestVV(dataRowNew[noAttr])
  if (closestVV.Val <= rowVVVals[noAttr])
    vvSmaller[noAttr] = closestVV
    vvSmallerDist[noAttr] = (rowVVVals[noAttr]) /
           attrRanges[noAttr] - vvSmaller[noAttr].Val
    vvGreater[noAttr] = vvSmaller[noAttr].Next
    if (vvGreater[noAttr] != null)
      vvGreaterDist[noAttr] = (vvGreater[noAttr].Val -
               rowVVVals[noAttr]) / attrRanges[noAttr]
  else
    vvGreater[noAttr] = closestVV
    if (vvGreater[noAttr] != null)
      vvGreaterDist[noAttr] = (vvGreater[noAttr].Val -
               rowVVVals[noAttr]) / attrRanges[noAttr]
smallerChosen = true
noCloseAttr = descendantNoAttrValsList[0].noAttr
if (vvSmallerDist[noCloseAttr] <=
vvGreaterDist[noCloseAttr])
  closestVV = vvSmaller[noCloseAttr]
else
  closestVV = vvGreater[noCloseAttr]
rowVVVal = rowVVVals[noCloseAttr]
ovRank = new List<OVRankEl>()
maxovRank = 0
do
  foreach (ov in closestVV.OVs)
    d = ov.AddDist((ov.VVList[noCloseAttr].Val - rowVVVal)
        / attrRanges[noCloseAttr], dataRowNew, attrRanges)
    if ((d >= 0)&& ((d <= maxovRank) || (ovRankCount < k)))
      i = ovRankCount - 1
      FindovRankPos(d, i)
      ovRank.Insert(i+1, new OVRankEl(d, ov))
      ovRankCount++
      if (ovRankCount > k)
        if (ovRank[k-1].d < ovRank[k].d)
          ovRank.RemoveRange(k, ovRankCount - k)
          ovRankCount = k
      maxovRank = ovRank.Last().d
  if (smallerChosen)
    vv = closestVV.Previous
    if (vv != null)
      vvSmaller[noCloseAttr] = vv
      vvSmallerDist[noCloseAttr] = (rowVVVals[noCloseAttr]
                      - vv.Val) / attrRanges[noCloseAttr]
    else
      vvSmaller[noCloseAttr] = null
  else
    vv = closestVV.Next
    if (vv != null)
      vvGreater[noCloseAttr] = vv
      vvGreaterDist[noCloseAttr] = (vv.Val -
         rowVVVals[noCloseAttr]) / attrRanges[noCloseAttr]
    else
      vvGreater[noCloseAttr] = null
  closestVVVal = Double.MaxVal
  for (i = 1; i < noAttrs; i++) // PartS
  {
    if (vvSmaller[i] != null)
      if (vvSmallerDist[i] < closestVVVal)
        closestVVVal = vvSmallerDist[i]
        smallerChosen = true
        noCloseAttr = i
    if (vvGreater[i] != null)
      if (vvGreaterDist[i] < closestVVVal)
        closestVVVal = vvGreaterDist[i]
        smallerChosen = false
        noCloseAttr = i
  if (closestVVVal == Double.MaxVal) break
  if (smallerChosen)
    closestVV = vvSmaller[noCloseAttr]
  else
    closestVV = vvGreater[noCloseAttr]
  rowVVVal = rowVVVals[noCloseAttr]
while ((closestVVVal <= maxovRank) || (ovRank.Count < k))
return GetWinningClass(ovRank)
```

The last algorithm uses rank tables (vvSmallerDist, vvGreaterDist) instead of rank list (vvRank) used in Algorithm 3 to move to the next smaller or greater closest value of the selected subset of the most invariant attributes. This mechanism allows to quickly update the pointers to the next smaller or next greater value for a given attribute using indices and immediate access to array elements instead of processing operations on the list (vvRank) as in Algorithm 3. Nevertheless, the search for the next closest value requires to loop through the values in these rank tables (PartS), whereas the rank list used in Algorithm 3 supplied us with such a closest value immediately. Hence, both approaches have their pros and cons, and only tests on datasets can determine which approach is the most efficient.

IV. Experimental Results and Comparisons

To prove that the associative search for k nearest neighbors (the k most similar objects in the collection) using AGDS structures is computationally more efficient, all four algorithms were run several times on the same benchmark datasets of different sizes and numbers of duplicates from ML Repository [12], and the average times were measured.

Table I shows that almost always the AKNN-T algorithm was the fastest, the second place was usually occupied by the AKNN-L algorithm, the third one by the AKNN-1 algorithm. Typically the slowest one was the classic KNN algorithm working on a table data.

The biggest increase in speed was achieved for Skin Data (Table 1) because these data contain many duplicated values for each attribute and have 193624 duplicated objects. In this data set, the aggregative properties of AGDS structures represent all these duplicates by single value vertices that can be searched through very fast. The other tested data also achieved the results a few times fasters, which proves that the proposed associative search approach to k nearest neighbors or similar objects search problem works. However, the proposed approach requires a little bit more programming effort than the classic approach based on the tabular structure. The results also show that independently on the number of attributes, the number of samples and the number of duplicates and data distribution in the dataspace always the AKNN-T algorithm (Algorithm 4) was the best in most of the cases (evaluated training datasets).

TABLE I. COMPARISONS OF KNN CLASSIFICATION EFFICIENCY OF THE CLASSIC AND ASSOCIATIVE APPROACHES USING AGDS STRUCTURE.

Datasets				Create	Time Efficiency of the Algorithms				Speed comparisons		
Name of dataset	Data Volume			AGDS	KNN	AKNN-1	AKNN-L	AKNN-T	AKNN-1	AKNN-L	AKNN-T
	Samples	Attributes	k	ms	ticks	ticks	ticks	ticks	to KNN	to KNN	to KNN
Immunotherapy	90	7	2	1	146	145	113	68	1.01	1.29	2.15
Immunotherapy	90	7	3	1	164	153	127	72	1.07	1.29	2.28
Immunotherapy	90	7	5	1	171	161	133	84	1.06	1.29	2.04
Immunotherapy	90	7	10	1	199	166	173	93	1.20	1.15	2.14
Immunotherapy	90	7	20	1	211	177	167	105	1.19	1.26	2.01
Iris	150	4	2	1	154	48	29	19	3.21	5.31	8.11
Iris	150	4	3	1	175	52	32	23	3.37	5.47	7.61
Iris	150	4	5	1	194	66	44	27	2.94	4.41	7.19
Iris	150	4	10	1	220	83	57	45	2.65	3.86	4.89
Iris	150	4	20	1	238	97	84	61	2.45	2.83	3.90
Banknote	1372	4	2	46	1349	161	240	106	8.38	5.62	12.73
Banknote	1372	4	3	46	1434	201	264	156	7.13	5.43	9.19
Banknote	1372	4	5	46	1469	289	376	187	5.08	3.91	7.86
Banknote	1372	4	10	46	1526	368	498	260	4.15	3.06	5.87
Banknote	1372	4	20	46	1569	497	680	340	3.16	2.31	4.61
Wine Quality Red	1599	11	2	69	4073	2372	1063	730	1.72	3.83	5.58
Wine Quality Red	1599	11	3	69	4137	2724	1229	837	1.52	3.37	4.94
Wine Quality Red	1599	11	5	69	4219	3265	1377	903	1.29	3.06	4.67
Wine Quality Red	1599	11	10	69	4296	3275	1646	1058	1.31	2.61	4.06
Wine Quality Red	1599	11	20	69	4422	3564	1794	1172	1.24	2.46	3.77
Skin Data	245057	3	2	114198	42952	743	467	439	57.81	91.97	97.84
Skin Data	245057	3	3	114198	43076	825	527	510	52.21	81.74	84.46
Skin Data	245057	3	5	114198	43856	934	573	617	46.96	76.54	71.08
Skin Data	245057	3	10	114198	44401	1332	761	826	33.33	58.35	53.75
Skin Data	245057	3	20	114198	45477	1682	1001	1064	27.04	45.43	42.74
Eye	14980	14	2	3142	56531	61733	10293	8982	0.92	5.49	6.29
Eye	14980	14	3	3142	56599	61778	10312	9015	0.92	5.49	6.28
Eye	14980	14	5	3142	57372	62312	11407	9720	0.92	5.03	5.90
Eye	14980	14	10	3142	57898	64150	12376	10411	0.90	4.68	5.56
Eye	14980	14	20	3142	58280	65844	13422	11633	0.89	4.34	5.01

Therefore, only the AKNN-T algorithm is used in the second experiment to find priorities (weights of the Sebestyen distance) for attributes of the datasets to achieve better classification results than when using Euclidean distance.

V. PRIORITIZING OF THE ATTRIBUTES AS HYPERPARAMETERS FOR OPTIMIZATION OF KNN PERFORMANCE

In the classic KNN approach, the input data space is used as it is for each attribute, and the chosen distance measure is computed, or all attribute values are normalized in order to avoid preference of the attributes defined by values of smaller ranges. Without normalization, the attributes in which values are defined in greater ranges than other attributes are automatically treated as less important during the search for k nearest neighbors because the distances of attributes of smaller ranges will be found as closer. To avoid this situation, all experiments made in this paper were performed on normalized datasets, but the normalizations were made on the fly during the runs of the algorithms. The use of normalized and non-normalized attribute values shows that the way how each attribute is treated during the computation of k nearest neighbors has a great impact on the search and classification results.

In the next experiments, we used prioritizing the attributes (i.e. the Sebestyen measure) to gain extra benefits from treating various attributes with different priorities. In this approach, each normalizing factor (previously determined by the range of all values of each attribute) was multiplied by the priority factor (weights), and these factors are different for various attributes. The priority factors (weights) were used as hyperparameters for KNN classifiers. These hyperparameters can be determined by various methods, e.g.: random search, mesh search, genetic algorithms, or evolutionary approaches. In our experiments, we used a random search because it was fast and sufficiently efficient to find sets of priorities of attributes for various datasets to improve classification results achieved by KNN classifiers.

Table II shows that the prioritizing (weighting) of attributes substantially improved classification results for all datasets, and it proves that attributes taken with various priorities have a positive impact on the classification process made by the KNN classifiers and methods searching for similar objects in the collection. All experiments were run on benchmark datasets where training data and testing data (dev data) were divided in ratio 90% to 10% of the volume, where testing data were chosen representatively due to the classes.

TABLE II. COMPARISONS OF KNN CLASSIFICATION RESULTS USING PRIORITIZING OF ATTRIBUTES TO THE CLASSIC APPROACH WITHOUT IT.

Datasets			Comparison Distances of the Best Classifications of AKNN-T				
Name of dataset	Data Volume		Euclidean		Sebestyen		Weights of the Attributes for Sebestyen Measure
	Samples	Attributes	Correct	k	Correct	k	
Immunotherapy	90	7	77.78%	3	100,00%	3	[2.674, 1.256, 0.649, 3.776, 1.912, 2.806, 3.436]
Iris	150	4	93.33%	3	100,00%	3	[0.651, 0.2353, 3.113, 0.825]
Banknote	1372	4	100.00%	3	100,00%	3	weighting unnecessary (all weights equal to 1)
Wine Quality Red	1599	11	58.75%	12	65.63%	28	[1.815, 2.451, 1.372, 0.817, 2.094, 2.810, 3.120, 2.820, 1.821, 0.871, 3.628]
Skin Data	245057	3	99.96%	3	99.96%	3	weighting unnecessary (all weights equal to 1)
Eye	14980	14	84.18%	3	85.85%	5	[1.172, 3.820, 2.254, 1.210, 2.885, 1.052, 0.878, 4.121, 2.581, 2.115, 2.528, 3.354, 0.206, 3.490]

VI. CONCLUSION

In this paper, we proposed three new algorithms based on an associative graph data structure that can accelerate the search of the most similar objects or nearest neighbors in comparison to the classic k nearest neighbors' approaches that use tables. The efficiency was achieved thanks to the associative data organization in the associative graph structures, i.e. aggregation or representation of duplicates and sorting all attribute values simultaneously.

We found that the algorithm using a rank table (Algorithm 4) and several most numerous number of unique values instead of only one attribute of the most numerous number of unique values had the best performance. We used the Sebestyen measure to set different priorities to attributes and achieve better performance than using Euclidean distance.

Finally, this paper presented that the use of associative data organization can benefit in various data processing tasks as demonstrated on classification or similar sample searches. The reason is that this organization allows us to look only through relevant objects (instead of all objects) to find answers to the asked questions. Hence, the associative representation of the relations between stored data can make the search routines more efficient, especially when considering big data sources, where looping through all objects is costly.

ACKNOWLEDGMENT

This work was supported by the grant from the National Science Centre, Poland DEC-2016/21/B/ST7/02220 and AGH 16.16.120.773.

REFERENCES

[1] T. Abidin and W. Perrizo, "A Fast and Scalable Nearest Neighbor Based Classifier for Data Mining," IEEE Global Congress on Intelligent Systems, pp. 536–540,2006.

[2] Jianping Gou, Lan Du, Yuhong Zhang and Taisong Xiong: "A New Distance-weighted k-nearest Neighbor Classifier," Journal of Information and Computational Science, vol. 9, no. 6, pp. 1429–1436, 2012.

[3] D. P. Vivencio, E. R. Hruschka, M. C. Nicoletti and E. B. Santos, "Feature-weighted k-Nearest Neighbor Classifier," IEEE Symposium on Foundations of Computational Intelligence, pp. 481–486,2007.

[4] J.W. Kalat, Biological grounds of psychology, Wadsworth Publishing, 2008.

[5] A. Horzyk, "Associative Graph Data Structures with an Efficient Access via AVB+trees," In 2018 11th International Conference on Human System Interaction, IEEE ,2018, pp. 169 - 175.

[6] A. Horzyk and K. Gołdon, "Associative Graph Data Structures Used for Acceleration of K Nearest Neighbor Classifiers," In 2018 27th International Conference on Artificial Neural Networks, 2018, pp. 648-658.

[7] A. Horzyk, Artificial Associative Systems and Associative Artificial Intelligence, EXIT, Warsaw, 2013.

[8] R. Agrawal, "Extensions of k-Nearest Neighbor Algorithm," Research Journal of Applied Sciences, Engineering and Technology, vol. 13, no. 1, pp. 24–29,2016.

[9] S. A. Dudani, "The distance-weighted k-nearest neighbor rule," IEEE Transactions on System, Man, and Cybernetics, vol. 6, pp. 325-327,1976.

[10] R. Jensen and Ch. Cornelis, "A New Approach to Fuzzy-Rough Nearest Neighbour Classification," International conference on rough sets and current trends in computing, 2008, pp. 310–319.

[11] I. H. Witten and E. Frank, Data mining: Practical machine learning tools and techniques, Morgan Kaufmann Publishers, 2005.

[12] UCI ML Repository, https://archive.ics.uci.edu/ml/index.php, last accessed 2019/07/25.