BANG-Clustering: A Novel Grid-Clustering Algorithm for Huge Data Sets

Erich Schikuta and Martin Erhart

Institute of Applied Computer Science and Information Systems, University of Vienna
Rathausstr. 19/4, A-1010 Vienna, AUSTRIA
schiki@ifs.univie.ac.at

Abstract. In this paper a new approach to hierarchical clustering of huge data sets is presented, which is based on a Grid-Clustering approach [Sch96]. It uses a multi-dimensional grid data structure, the BANG-structure, to organize the value space surrounding the pattern values. The patterns are grouped into blocks and clustered with respect to the blocks by a topological neighbor search algorithm.

1 Introduction

Conventional clustering algorithms show serious problems in clustering large pattern sets, due to hard-to-meet run-time and memory requirements of the underlying algorithms. This led to the development of alternative methods, which cluster the patterns according to the structure of the embedding space (see [WK79], [Bro90], or [Sch96]). Based on this approach the BANG-Clustering algorithm presented in this paper uses the block information of a modified multi-dimensional BANG-file structure and clusters the patterns accordingly to their surrounding blocks creating a respective dendrogram in turn.

According to the clustering algorithm classification presented by Dubes et al. [DJ80] BANG-Clustering can be identified as

- agglomerative - small clusters are combined to larger clusters,
- exclusive - clusters are not disjoint (i.e. non overlapping),
- intrinsic - only pattern information is used,
- hierarchical - produces a nested clustering, represented as dendrograms,
- polythetic - uses all pattern features, and
- exhaustive - all patterns are clustered.

2 The BANG-Structure

The BANG-Structure stores the patterns of the underlying value space by a grid structure, which is called *grid directory* (similarly to the Grid-File). This structure (see figure 1), which is administrated by *scales*, partitions the k-dimensional value space into *grid regions* (rectangular shaped subspaces). Each scale represents one pattern attribute, and each scale entry resembles a (k-1) dimensional hyperspace splitting the value space into two.

Each grid region is mapped to one *data block* containing the patterns, but a data block can be mapped by more than one grid region (1:m mapping). The union of these grid regions (mapping to the same data block) is called *block region*. The value space spanned by a block region is rectangular shaped (convex).

2.1 Representation of the value space

The value space is partitioned into a *hierarchical* set of grid regions. Each region is uniquely identified by a pair of keys (r, l), where r is the region number and l is the level number.

The partitioning is binary (a region is split into two equally shaped regions) in each dimension. The sequence of the split dimensions has to be uniquely defined. An example is shown in figure 2. Region (0,0) comprises the whole value space and is partitioned according the defined scheme into subregions.

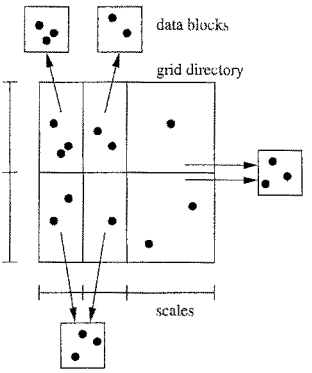

Fig. 1. BANG-Structure

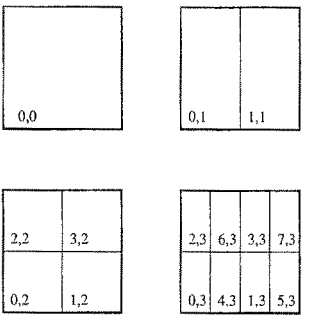

Fig. 2. Partitioning scheme

Contrary to the Grid-File a grid entry in the grid directory of the BANG-Structure maps to only a *single* data block. This means that only block regions are administrated, which results in a smaller grid directory than in the Grid-File. The structure of the block regions is defined by the following two axioms [Fre87]

- The union of all subregions into which the value space has been partitioned must span the whole value space.
- If two subregions intersect, then one of these subregions completely encloses the other.

The second axiom allows nested regions (contrary to the Grid-File), which is shown in figure 3. To reach compact structures algorithms are defined [Fre87], which guarantee a balance between the data blocks by redistribution. This proved extremely useful for clustered value sets.

2.2 Mapping scheme

The pattern values are transformed to the scale (0..1) in each dimension. Each pattern is mapped into its respective region. The region is defined by the pattern values in each dimension accordingly to the scales. The pattern value is p_i and the extent of value space l_i in dimension i. The scale value d_{i,l_i} is defined by

$$d_{i,l_i} = \lfloor p_i * 2^{l_i} \rfloor, \tag{1}$$

and the scale value of a higher level j_i is calculated simply from d_{i,l_i} by

$$d_{i,j_i} = \left\lfloor \frac{d_{i,l_i}}{2^{l_i - j_i}} \right\rfloor. \tag{2}$$

Based on this numbering scheme a value mapping function can be defined by the following rules:

- Level l+1 is created by splitting each region in a specific dimension.
- The number of regions of the splitting level l is 2^l.
- The region (r,l) is split into (r,l+1) and (r+2^l, l+1).
- Region number r and r+2^l are two possible extensions of the binary representation of r by one bit (the most significant).
- The value space of a scale is doubled, if the splitting level in dimension i increases from l_i to l_i+1. This represents an extension by one bit.

Derived form these rules it is obvious that in region 0, level 0, i.e. (0,0), each region of level l can be represented by l bits (l > 0) in binary. A function bits(p) can be defined, which returns the minimal number of bits of the value p by

$$bits(d_{i,l_i}) = l_i, \forall l_i \geq 0. \tag{3}$$

Because of the situation that the sum of all sub-levels comprises the whole value space, i.e.

$$l = \sum_{i=1}^{n} l_i, \forall l_i \geq 0, \tag{4}$$

it concludes that

$$bits(r) = \sum_{i=1}^{n} bits(d_{i,l_i} - l) = l, \forall l_i \geq 0. \tag{5}$$

The mapping of the pattern values to a unique region number can now be performed by a simple concatenation of the binary representation of the scale values. Sub-levels with the domain 0 are neglected. The split of a dimension doubles the number of grid regions (i.e. [r, l+1] and [r*2^l, l+1]). This is represented by adding one bit (the most significant one) in the region number and another bit (the least significant one) in the sub-level of the dimension in focus. By this algorithm the sequence of splits is fixed, in our special case it cyclic (the domain of each dimension is between 0 and 1). This algorithm is depicted by the *Mapping*-algorithm.

Mapping-algorithm

```
r=0; offset = 1; k = 0;
while (k < l) {
  i = k mod n + 1;
  j = k div n + 1;
  if (l[i] >= j) {
    r = r + offset * b[i, l[i] - j];
    offset = offset * 2;
    k = k + 1;
  }
}
```

2.3 Logical Regions

As shown in figure 4 a *logical region* is build by a region comprising a given value space (R1 in the figure) *minus* the value space of all regions contained in this region (R2, R3, and R4 in the figure).

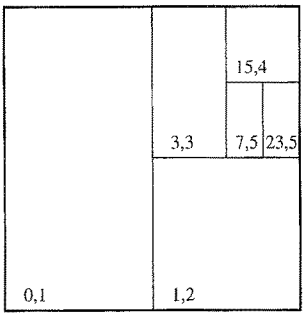

Fig. 3. BANG-Structure block regions

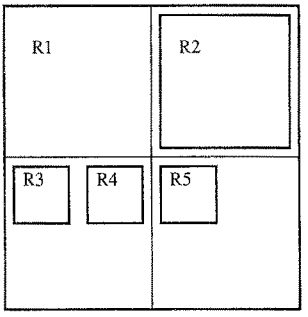

Fig. 4. Logical regions

Generally each logical region is defined by a set of regions so that a (convex) region contains the other regions, which in turn subtract from the first region. The resulting space can be concave. The block regions are stored in the grid directory. The data blocks contain the patterns of the respective logical regions. All balancing is done on the logical regions. The directory contains unique binary tuples of the block regions. By sorting these tuples according to the levels no ambiguity exists. By the mapping function always the smallest (largest by numbers) levels is found. This tuple is searched in the directory. If it is found the algorithm finishes, otherwise the region is searched which comprises the respective grid region.[1] Therefore, though the directory contains no information about

[1] Obviously the algorithm stops in region (0, 0) which comprises everything.

the logical regions, the correct unique mapping of the tuples to the regions is guaranteed.

2.4 Splitting and Merging

Splitting of a region is necessary if a tuple is inserted into a full region (data bucket). The logical region is split into two regions - one region comprises the other one - and the region containing more tuples is split iteratively until balance is reached.

Merging is much simpler than for the Grid File due to lack of deadlocks. If the number of tuples goes under a defined threshold a merging with the buddy is tried. If there is no buddy the region is merged with the comprising region. If the tuple number after merging exceeds the region limit a subsequent split has to be done, which yields a better tuple distribution.

3 BANG-Clustering

The BANG-Clustering algorithm uses the block information of the grid directory and clusters the patterns to their blocks accordingly.

3.1 Density Index

The algorithm calculates a *density index* of each block via the numbers of patterns and the spatial volume of the block. The spatial volume V_B of a block B is the Cartesian product of the extents e of block B in each dimension, i.e.

$$V_B = \prod e_{B_i}, i = 1, ..., k. \tag{6}$$

The density index D_B of block B is defined as the ratio of the actual number of patterns p_B contained in block B to the spatial volume V_B of B, i.e.

$$D_B = \frac{p_B}{V_B} \tag{7}$$

The blocks are sorted accordingly to their density indices. Blocks with the highest density index (obviously with highest pattern correlation) become clustering centers. The remaining blocks are then clustered iteratively in order of their density index, thereby building new cluster centers or merging with existing clusters. Only blocks adjacent to a cluster, i.e. *neighbors*, can be merged.

3.2 Neighbors

Two types of neighborhood can be distinguished in the BANG-Structure, *normal neighborhood*, i.e. neighbors respective block regions, and *refined neighborhood*, i.e. neighbors respective logical regions. Further a *neighbor degree* can be defined by the dimensionality of the "touching" area between 2 regions. Generally

the dimensionality can vary between 0 (an point) and k-1 (a k-1 dimensional hyperplane). For the example shown in figure 5 (2-dimensional case) the level of dimensionality is 0 (a point) and 1 (an edge). A normal neighborhood exists e.g. between regions R2 and R1, R3, R6, and R7, and a refined neighborhood between regions R2 and R1, R6, and R7.

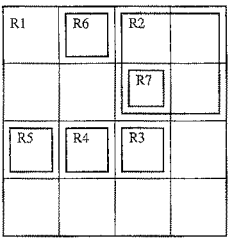

Fig. 5. Grid directory, GD

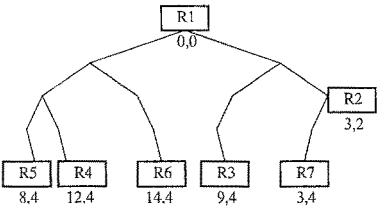

Fig. 6. GD figure 5 as binary tree

Neighbors are found by comparison of the scale values of the grid directory (see the Neighbor-algorithm). If regions are at the same level, the differences can be determined directly. If the levels are not at the same level, the lower level region has to be transformed to the higher level region and the comparison has to be done appropriately. In the example of figure 6 the regions and their identifiers are R1 = (0,0), R2 = (3,2), R3 = (9,4), R4 = (12,4), R5 = (8,4), R6 = (14,4), and R7 = (3,4). R6 and R2 are neighbors but on different region-levels (R6 on level 4, with x =1 and y = 3, and R2 on level 2, with x =1 and y = 1). Therefore we have to transform R2 to level 4, which yields in $x_{min} = 2$, $x_{max} = 3$, and $y_{min} = 2$, $y_{max} = 3$.

The region identifier are an ordered set of tuples. To find possible neighbor regions the algorithm accesses these tuples. To support this step efficiently we designed a novel administration structure for the region identifiers. Because of the numbering scheme of the grid regions we chose a binary tree for storing the grid structure. The basic scheme is shown in figure 6. The split of a region is directly supported by the tree scheme, and a comprising region is simply found by backtracking the path to the root (representing the whole value space). The height of the tree is defined by the number of region levels.

3.3 Dendrogram

The dendrogram is calculated directly by the clustering algorithm. The density indices of all regions are calculated and sorted in decreasing order. Starting with the first region (with the highest density index) all neighbor regions are determined and classified in decreasing order (step 1). The neighbor search is repeated for each processed region. The found regions are placed in the dendrogram to the right of the original regions (step 2), respective to the following rules,

Neighbor-algorithm

```
neighbor(region1, region2) {
  AllowedCount = ... // to be specified;
  // calculate grid values
  UnmapRegion(region1, grid);
  UnmapRegion(region2, GridCmp);
  if (levels of region1 and region2 are equal) {
    count = 0;
    for (each dimension d) {
      diff = abs(grid[d] - GridCmp[d]);
      if (diff > 1) break; // no neighbor
      else count = count + 1;
    }
    if (count > AllowedCount) { ... // this is no neighbor }
  } else {
    // transform higher to lower level - result in GridMin and GridMax
    TransformLevel(region1, region2);
    count = 0;
    for (each dimension d)
      // check that cmp[d] is in range [GridMin[d]-1, GridMax[d]+1]
    if (count > AllowedCount) { ... // this is no neighbor }
  }
}
```

- is R1 neighbor of R2 and R2 neighbor of R3 and R1 > R2 > R3, then build with R1, R2, and R3 a cluster (neighbor search starting from R3), and
- is R1 neighbor of R2 and R2 neighbor of R3 and R1 > R2 < R3, then build with R1, R2, and R3 a cluster (neighbor search starting from R2).

4 The BANG-Clustering System

We implemented the BANG-Clustering algorithm under the Unix operating system using X11 and the Motif libraries providing a user friendly environment, named the BANG-Clustering System (see [SE97]). Running versions are available for Linux, Sun and HP workstations.

Compared in the performance to conventional hierarchical and partitional algorithms and the Grid-Clustering method as well BANG-Clustering excelled all these methods. It is capable to analyze data sets, which were previously not tractable due to their size and/or dimensionality without any supplemental input information. A more comprehensive comparison of the performance results can be found in [SE97].

5 Summary

In this paper we presented BANG-Clustering, a hierarchical clustering method, which is based on the idea to organizes the pattern set space, which is partitioned

Dendrogram-algorithm

```
dendrogram() {
  // we build two lists:
  //   DendroList, contains the region with the highest density
  //   RestOfRegions, contains all other regions
  DendroList = FirstRegion;
  for (all regions x in DendroList) {
    // now we are looking for all neighbors in RestOfRegions
    FindNeighbors(x, RestOfRegions);
  }
}

FindNeighbors(region, SetOfRegions) {
  for (all regions x in SetOfRegions) {
    if (neighbor(region, x)) {
    // if a region is identified as a neighbor, it is inserted acc.
    //   to the density index, starting from the position of region x
      InsertIntoDendroList(x);
    }
  }
}
```

by a multi-dimensional data structure. This method is an extension of the Grid-Clustering algorithm presented in [Sch96], and is capable to cluster pattern sets of sizes far beyond the limits of conventional methods.

References

[Bro90] A.J. Broder. Strategies for efficient incremental nearest neighbor search. *Pattern Recognition*, 23:171–178, 1990.

[DJ80] R. Dubes and A.K. Jain. *Clustering methodologies in exploratory data analysis*, volume 19, pages 113–228. Academia Press, 1980.

[Fre87] M.W. Freestone. The bang file: A new kind of grid file. In *Proc. Special Interest Group on Management of Data*, pages 260–269. ACM, May 1987.

[Sch96] E. Schikuta. Grid clustering: An efficient hierarchical clustering method for very large data sets. In *Proc. 13th Int. Conf. on Pattern Recognition*, volume 2, pages 101–105. IEEE Computer Society, 1996.

[SE97] Erich Schikuta and Martin Erhart. The bang-clustering system: Grid-based data analysis. In X Liu, P. Cohen, and M. Berthold, editors, *Advances in Intelligent Data Analysis. reasoning about Data, Proc. Second International Symposium IDA-97*, volume 1280 of *LNCS*, London, UK, August 1997. Springer-Verlag.

[WK79] C.S. Warnekar and G. Krishna. A heuristic clustering algorithm using union of overlapping pattern-cells. *Pattern Recognition*, 11:85–93, 1979.