An Information-Theoretical Approach to Genetic Algorithms for Clustering

Dana Cristofor Dan A. Simovici*

June 14, 2001

Abstract

In a database with categorical attributes each attribute defines a partition whose classes can be regarded as natural clusters of rows. The main theme of this paper is finding a partition of the rows of the database that is as close as possible to the partitions associated to each attribute. The classes of this partition will then be treated as clusters of rows. We evaluate the closeness of two partitions by using certain generalizations of the classical conditional entropy. From this perspective, we wish to construct a partition (referred to as the median partition) such that the sum of the dissimilarities between this partition and all the partitions determined by the attributes of the database is minimal. Then, the problem of finding the median partition is an optimization problem over the space of all partitions of the rows of the database for which we give an approximative solution. To search more efficiently the space of possible partitions, which can be very large, we are using a genetic algorithm. Partitions are represented by chromosomes and we tested both the classical techniques of mutation and crossover and certain special mutation and crossover methods that contain specific knowledge of the problem domain.

Keywords: median partition, Shannon entropy, Gini index, mutations, crossover operations

1 Introduction

Clustering is a main topic of computer science and is particularly important in the field of data mining. Research of this problem has resulted in many clustering algorithms: [ABKS99], [AGGR98], [ChWcY99], [MHPJX96], [SRK99], [HGV97], [TRL96], [SCC00a]. For an overview of clustering algorithms see [Mir96], [KR90], and [JMF99].

In this paper we investigate the application of information-theoretical methods to genetic algorithms for clustering. The objects that we cluster are represented by tuples in a table and the goal of the genetic algorithms is to construct

*University of Massachusetts at Boston, Department of Mathematics and Computer Science, Boston, Massachusetts 02125, USA, `e-mail: {dana, dsim}cs.umb.edu`

a partition of the set of rows whose classes will be regarded as clusters of the rows. We will require that this partition satisfy the following two conditions:

1. it must summarize the partitions generated by the attributes of the table.
2. the number of classes of the partition should not exceed a prescribed upper limit.

The first condition ensures that the resulting partition represents a clustering of the rows, while the second condition represents a restriction of the number of classes in this clustering.

In literature, the partition satisfying the first condition is called *median partition* and was analyzed in [BL95], [Rég83a] and [Rég83b]. Research concerning the application of genetic algorithms to grouping problems was presented in [Mic99], [Fal99], and [JB91].

Finding the median partition is an NP-complete problem as it was shown by [BL95]. That is why we focus in this paper on finding an approximative solution using genetic algorithms. In order to determine how well a partition summarizes the attribute partitions of a table we will use a measure based on a generalization of the entropy that was introduced in [SCC00c] and [SCC00b].

The rest of the paper is organized as follows: Section 2 introduces definitions and notation, and presents a generalization of the classical entropy measure and important properties of this generalization. Section 3 describes the genetic algorithms that we will use to search for the median partition. Section 4 presents the results of our experiments as well as an analysis of the implications of the different techniques used. Section 5 concludes the paper with a summarization of our results and a discussion of future research topics.

2 Definitions. Generalized entropy

The set of partitions of a set R is denoted by $\mathsf{PART}(R)$. For $\pi, \sigma \in \mathsf{PART}(R)$ we write $\pi \leq \sigma$ if every class of σ is a union of classes of π. We will denote by ω_R the partition with one class, containing all the elements of R and by α_R the partition will $|R|$ classes, each class containing one element of the set R.

Let $\mathcal{U}$ be a set whose elements are referred to as *attributes*. We assume that for every element A of $\mathcal{U}$ there is a set denoted by $\mathsf{Dom}(A)$ referred to as the *domain of* A such that $|\mathsf{Dom}(A)| \geq 2$.

Definition 2.1 A *table* is a function $T : R \times H \longrightarrow \bigcup D_H$, where R, a finite set, is referred to as the *set of rows of* T, H is a finite subset of $\mathcal{U}$ (called the *heading of* T), $D_H = \{\mathsf{Dom}(A) \mid A \in H\}$, and $T(r, A) \in \mathsf{Dom}(A)$ for every $r \in R$ and $A \in H$.

The *projection of the table* $T : R \times H \longrightarrow \bigcup D_H$ *on the set of attributes* $L \subseteq H$ is the table $T[L] : R \times L \longrightarrow \bigcup D_L$ given by $T[L](r, A) = T(r, A)$ for every $r \in R$ and $A \in L$. If r is a row of the table T, then the set $\{T(r, A) \mid A \in H\}$ is the *content* of the row r. Note that this definition of a table allows for the existence of multiple rows having the same content.

The *projection of the row r of the table T on L* is the set $r[L] = \{T(r, A) \mid A \in L\}$. For $A \in H$, $Q_T[A] = \{T(r, A) \mid r \in R\}$ represents the content of the column associated with the attribute A in the table T.

The *active domain of a set of attributes L in the table T* is the set $\mathsf{aDom}_T(L) = r[L]$. The cardinality of the active domain of L in the table T will be denoted by a_L^T, or, when there is no risk for confusion, just by a_L.

Let $T : R \times H \longrightarrow \bigcup D_H$ be a table. Every attribute set $L \subseteq H$ determines a partition $\pi_L = \{D_1, \ldots, D_{a_L}\}$ of the set R of rows in table T, where $\mathsf{aDom}_T(L) = \{v_1, \ldots, v_{a_L}\}$ and

$$D_i = \{r \in R \mid T(r, L) = v_i\}$$

for $1 \leq i \leq a_L$. Clearly, if $L = \{A_{i_1}, \ldots, A_{i_\ell}\}$, then $\pi_L = \bigcap\{\pi_{A_{i_j}} \mid 1 \leq j \leq \ell\}$.

Let $\mathbb{R}$ be the set of reals. The k-dimensional simplex is the set

$$\mathsf{SIMPLEX}_{k-1} = \{(p_1, \ldots, p_k) \in \mathbb{R}^k \mid p_i \geq 0 \text{ and } p_1 + \cdots + p_k = 1\}.$$

A function $f : \mathbb{R} \longrightarrow \mathbb{R}$ is *concave on a set $S \subseteq \mathbb{R}$* if $f(\alpha x + (1 - \alpha)y) \geq \alpha f(x) + (1 - \alpha)f(y)$ for $\alpha \in [0, 1]$ and $x, y \in S$. The function f is *sub-additive* (*supra-additive*) on S if $f(x + y) \leq f(x) + f(y)$ ($f(x + y) \geq f(x) + f(y)$) for $x, y \in S$.

Definition 2.2 A *generator* is a concave, subadditive function $f : [0, 1] \longrightarrow \mathbb{R}$ such that $f(0) = f(1) = 0$ and

$$f(\theta a_1) + \cdots + f(\theta a_n) \leq \theta(f(a_1) + \cdots + f(a_n)) + f(\theta). \tag{1}$$

for every $(a_1, \ldots, a_n) \in \mathsf{SIMPLEX}_{n-1}$ and $\theta \in [0, 1]$.

The impurity measure induced by f is

$$i(p_1, \ldots, p_k) = f(p_1) + \cdots + f(p_k),$$

for every $(p_1, \ldots, p_k) \in \mathsf{SIMPLEX}_{k-1}$.

It is easy to verify that such functions as $f_{\text{gini}}(p) = p - p^2$ (the Gini index), $f_{\text{sq}}(p) = \sqrt{p} - p$, $f_{\text{ent}}(p) = -p \log p$ (the Shannon entropy), or f_{peak} (given by $f_{\text{peak}}(p) = p$ for $0 \leq p \leq 0.5$ and $f_{\text{peak}}(p) = 1 - p$ for $0.5 < p \leq 1$) are generators.

Definition 2.3 Let f be a generator, R be the set of rows of a table T, and let $\pi = \{B_1, \ldots, B_n\}$, $\sigma = \{C_1, \ldots, C_m\}$ be two partitions of R.

The *f-impurity of partition π* (called also the *f-entropy* of π) is $\mathcal{H}^f(\pi) = \sum_{i=1}^n f\left(\frac{|B_i|}{|R|}\right)$.

The *specific f-impurity of a subset L of R relative to a partition π* is the quantity

$$\mathsf{imp}_\pi^f(L) = f\left(\frac{|L \cap B_1|}{|L|}\right) + \cdots + f\left(\frac{|L \cap B_n|}{|L|}\right).$$

The f*-impurity of* π *relative to* σ is

$$\mathcal{H}^f(\pi|\sigma) = \sum_{j=1}^{m} \frac{|C_j|}{|R|} \mathsf{imp}_\pi^f(C_j) = \frac{1}{|R|} \sum_{j=1}^{m} |C_j| \sum_{i=1}^{n} f\left(\frac{|B_i \cap C_j|}{|C_j|}\right).$$

This quantity is also called the f*-conditional entropy of* π *relative to* σ.

In other words, the conditional entropy $\mathcal{H}^f(\pi|\sigma)$ is the average value of the specific impurity of the classes of the partition σ relative to the partition π. Note that $\mathcal{H}^f(\pi|\omega_R) = \mathcal{H}^f(\pi)$ for every $\pi \in \mathsf{PART}(R)$.

We have $\sigma \leq \pi$, if and only if $\mathcal{H}^f(\pi|\sigma) = 0$. Thus, if α_R is the least partition of a set R and ω_R is the largest such partition, then, for any partition of the set R we have $\mathcal{H}^f(\pi|\alpha_R) = 0$ and $\mathcal{H}^f(\omega_R|\pi) = 0$.

Theorem 2.4 *Let* f *be a generator and let* π, σ *be two partitions of a set* R. *We have:*

$$\mathcal{H}^f(\pi) \geq \mathcal{H}^f(\pi|\sigma) \geq \mathcal{H}^f(\pi \wedge \sigma) - \mathcal{H}^f(\sigma).$$

Proof. Suppose that $\pi = \{B_1, \ldots, B_n\}$ and $\sigma = \{C_1, \ldots, C_m\}$. To prove the first inequality observe that we can write

$$\frac{|B_i|}{|R|} = \sum_{j=1}^{m} c_j \frac{|B_i \cap C_j|}{|C_j|},$$

where $c_j = \frac{|C_j|}{|R|}$ for $1 \leq j \leq m$ and $(c_1, \ldots, c_m) \in \mathsf{SIMPLEX}_{m-1}$. The concavity of f implies

$$f\left(\frac{|B_i|}{|R|}\right) \geq \sum_{j=1}^{m} c_j f\left(\frac{|B_i \cap C_j|}{|C_j|}\right),$$

which, in turn, yields

$$\mathcal{H}^f(\pi|\sigma) = \sum_{i=1}^{n} \sum_{j=1}^{m} c_j f\left(\frac{|B_i \cap C_j|}{|C_j|}\right) \leq \sum_{i=1}^{n} f\left(\frac{|B_i|}{|R|}\right) = \mathcal{H}^f(\pi).$$

For a fixed j let $b_i = \frac{|B_i \cap C_j|}{|C_j|}$ and $\theta = \frac{|C_j|}{|R|}$. Since $(b_1, \ldots, b_n) \in \mathsf{SIMPLEX}_{n-1}$ we have

$$\sum_{i=1}^{n} f\left(\frac{|B_i \cap C_j|}{|R|}\right) \leq \frac{|C_j|}{|R|} \sum_{i=1}^{n} f\left(\frac{|B_i \cap C_j|}{|C_j|}\right) + f\left(\frac{|C_j|}{|R|}\right).$$

Summing these inequalities for $1 \leq j \leq m$ and observing that the classes of the partition $\pi \wedge \sigma$ have the form $B_i \cap C_j$ we obtain the second inequality of the theorem. ∎

Theorem 2.5 *If $\pi_1, \pi_2, \sigma \in \mathsf{PART}(M)$ are such that $\pi_1 \leq \pi_2$, then $\mathcal{H}^f(\pi_1|\sigma) \geq \mathcal{H}^f(\pi_2|\sigma)$.*

If $\pi, \sigma_1, \sigma_2 \in \mathsf{PART}(R)$ are such that $\sigma_1 \leq \sigma_2$, then $\mathcal{H}^f(\pi|\sigma_1) \leq \mathcal{H}^f(\pi|\sigma_2)$.

Proof. Let $\pi_1, \pi_2 \in \mathsf{PART}(M)$ be such that $\pi_1 \leq \pi_2$. If $\pi_2 = \{B_1, \ldots, B_n\}$, then for each class B_i of π_2 there exists a set of classes $\mathfrak{B}_i = \{B_{i1}, \ldots, B_{ik_i}\}$ of π_1 such that $B_i = B_{i1} \cup \cdots \cup B_{ik_i}$. Therefore, $B_i \cap C_j = \bigcup\{B_{i\ell} \cap C_j | 1 \leq \ell \leq k_i\}$, and the subadditivity of f implies

$$f\left(\frac{|B_i \cap C_j|}{|C_j|}\right) \leq \sum_{\ell=1}^{k_i} f\left(\frac{|B_{i\ell} \cap C_j|}{|C_j|}\right).$$

In turn, this implies $\mathcal{H}^f(\pi_1|\sigma) \geq \mathcal{H}^f(\pi_2|\sigma)$.

Let now $\sigma_1, \sigma_2 \in \mathsf{PART}(R)$ be two partitions such that $\sigma_1 \leq \sigma_2$. Denote by $\mathfrak{C}_j$ the subset of σ_1 that contains the classes included in the class C_j of σ_2, where $\mathfrak{C}_j = \{C_{j1}, \ldots, C_{jh_j}\}$. Note that we can write

$$\frac{|B_i \cap C_j|}{|C_j|} = c_1 \frac{|B_i \cap C_{j1}|}{|C_{j1}|} + \cdots + c_{h_j} \frac{|B_i \cap C_{jh_j}|}{|C_{jh_j}|},$$

where $c_p = \frac{|C_{jp}|}{|C_j|}$ for $1 \leq p \leq h_j$. The concavity of the function f implies

$$f\left(\frac{|B_i \cap C_j|}{|C_j|}\right) \geq \sum_{p=1}^{h_j} \frac{|C_{jp}|}{|C_j|} \cdot f\left(\frac{|B_i \cap C_{jp}|}{|C_{jp}|}\right),$$

which implies the second inequality of the theorem. ∎

Corollary 2.6 *If $\pi_1, \pi_2, \sigma \in \mathsf{PART}(M)$ are such that $\pi_1 \leq \pi_2$, then $\mathcal{H}^f(\pi_1) \geq \mathcal{H}^f(\pi_2)$.*

Proof. This statement follows from Theorem 2.5 by taking $\sigma = \omega_M$. ∎

Corollary 2.7 *For all partitions $\pi_1, \ldots, \pi_n \in \mathsf{PART}(R)$ we have*

$$\frac{1}{n} \sum_{i=1}^{n} \mathcal{H}^f(\pi_i) \leq \mathcal{H}^f(\bigwedge_{i=1}^{n} \pi_i) \leq \sum_{i=1}^{n} \mathcal{H}^f(\pi_i).$$

Proof. The first inequality follows from Corollary 2.6; the second follows from Theorem 2.4. ∎

Theorem 2.8 *Let $\pi_1, \pi_2, \sigma \in \mathsf{PART}(M)$ and let f be a generator. Then,*

$$\mathcal{H}^f(\pi_1 \wedge \pi_2|\sigma) \leq \mathcal{H}^f(\pi_1|\pi_2 \wedge \sigma) + \mathcal{H}^f(\pi_2|\sigma).$$

Proof. Suppose that $\pi_1 = \{B_1, \ldots, B_n\}$, $\pi_2 = \{D_1, \ldots, D_k\}$, and $\sigma = \{C_1, \ldots, C_m\}$. Applying the property of generators described by Equation (1) for

$$\left(\frac{|B_1 \cap D_\ell \cap C_j|}{|D_\ell \cap C_j|}, \ldots, \frac{|B_n \cap D_\ell \cap C_j|}{|D_\ell \cap C_j|}\right) \in \mathsf{SIMPLEX}_{n-1}$$

and $\theta = \frac{|D_\ell \cap C_j|}{|C_j|}$, we obtain the inequalities:

$$\sum_{i=1}^{n} f\left(\frac{|B_i \cap D_\ell \cap C_j|}{|C_j|}\right) \leq \frac{|D_\ell \cap C_j|}{|C_j|} \cdot \sum_{i=1}^{n} f\left(\frac{|B_i \cap D_\ell \cap C_j|}{|D_\ell \cap C_j|}\right) + f\left(\frac{|D_\ell \cap C_j|}{|C_j|}\right)$$

for $1 \leq \ell \leq k$ and $1 \leq j \leq m$. Summing these inequalities for ℓ and j yields the desired inequality since the classes of the partition $\pi_2 \wedge \sigma$ have the form $D_\ell \cap C_j$. ∎

Corollary 2.9 *Let $\pi_1, \ldots, \pi_n, \sigma \in \mathsf{PART}(M)$ and let f be a generator. Then,*

$$\mathcal{H}^f\left(\bigwedge_{i=1}^{n} \pi_i | \sigma\right) \leq \sum_{i=1}^{n} \mathcal{H}^f\left(\pi_i | \bigwedge_{j=i+1}^{n} \pi_j \wedge \sigma\right).$$

Proof. The inequality of the corollary follows immediately from Theorem 2.5. ∎

A generalization of the notion of metric on a set is the notion of *dissimilarity* (see[KR90] and [JMF99]) which is useful in clustering.

Definition 2.10 A *dissimilarity on a set S* is a function $d : S \times S \longrightarrow \mathbb{R}$ that satisfies the following conditions:

1. $d(x, y) \geq 0$ and $d(x, x) = 0$;
2. $d(x, y) = d(y, x)$,

for every $x, y \in S$.

The dissimilarity d is said to be *definite* if for all $x, y \in S$, $d(x, y) = 0$ implies $x = y$. If a dissimilarity satisfies the triangular inequality $d(x, y) + d(y, z) \geq d(x, z)$ for $x, y, z \in S$, then we obtain the familiar notion of metric on S.

If f is a generator, then the mapping $d^f : \mathsf{PART}(R) \times \mathsf{PART}(R) \longrightarrow \mathbb{R}$ given by

$$d^f(\pi, \sigma) = \mathcal{H}^f(\pi|\sigma) + \mathcal{H}^f(\sigma|\pi)$$

for $\sigma, \pi \in \mathsf{PART}(R)$, is clearly a definite dissimilarity on $\mathsf{PART}(R)$. When π is close to σ, meaning that their classes have many elements in common, then both $\mathcal{H}^f(\pi|\sigma)$ and $\mathcal{H}^f(\sigma|\pi)$ are close to 0, so $d^f(\pi, \sigma)$ is close to 0.

3 Clustering using Genetic Algorithms

Let $T : R \times H \longrightarrow \bigcup D_H$ be a table. Our goal is to find a partition of the set of rows R (that is a clustering of the rows of R) that has no more classes that a certain prescribed limit and summarizes the partitions determined by all the attributes $A \in H$. This is achieved by finding a partition π^* such that

1. $|\pi^*| \leq k$, and
2. π^* minimizes the sum $\sum_{A \in H} d^f(\pi_A, \pi^*)$.

For each attribute $A \in H$ we define a vector $C_A = Q_T[A]$ containing the values of the attribute A as they appear if we look on the column A of the table T. Since the value of the entropy of a partition depends only on the classes of the partition and their cardinalities and not on the particular values of the elements belonging to these classes, we will use a numerical encoding of the distinct values in the active domain of the attributes. Similarly, the value of the conditional entropy between two partitions depends only on intersection patterns of the classes of the two partitions; specifically, the conditional entropy depends on the cardinality of the classes and the cardinality of the intersections between classes. Therefore, the above encoding does not influence the result of the entropy. Thus, in the encoding that we use for each distinct value that appears in column A, we will assign a unique number between 1 and $|\pi_A|$.

We call an *almost-median partition* the partition discovered by a genetic algorithm in its search for the median partition. We introduce this notion since it is not guaranteed that a genetic algorithm will find the median partition, it is possible that it will terminate after finding an approximation of it.

Definition 3.1 A *k-chromosome* on a table $T : R \times H \longrightarrow \bigcup D_H$ is a function $K : R \longrightarrow \{1, \dots, k\}$. An element of the set $\{1, \dots, k\}$ is called a *class identifier.*

The partition π_K of the set of rows R determined by the k-chromosome K is $\pi_K = \{C_1, \dots, C_k\}$, where $C_j = \{r \in R \mid K(r) = j\}$ for $1 \leq j \leq k$.

The targeted number of classes of the partition π^* is an input parameter k specified by the user. The chromosomial population of the algorithm consists of k-chromosomes, $K_1, K_2 \dots K_M$, where M is another input parameter. Each k-chromosome K_i can be regarded as a sequence of length $N = |R|$ representing a possible assignment of the rows of the table T to the k classes of the partition of the chromosome K_i. Initially, the k-chromosomes $K_1, K_2 \dots K_M$ are generated using random values between 1 and k. For example, if $N = 6$, $k = 2$ and K_1 was initialized with [1 2 1 1 2 2], it means that the first, third and fourth row of the database belong to the first class, while the second, fifth and sixth row of the table belong to the second class of the partition defined by K_1.

The idea of the genetic evolution is to modify the chromosomes in the current population by using mutation and crossover as genetic operators such that in the new population we will have chromosomes that will be increasingly closer to the target median partition, that is, they will summarize better and better the columns of the table T.

To introduce fitness measures for chromosomes used in clustering of rows of table $T : R \times H \longrightarrow \bigcup D_H$ we define:

$$\begin{aligned}\mathcal{K}^f(\pi) &= \sum_{A \in H} \mathcal{H}^f(\pi_A|\pi)\\ \mathcal{L}^f(\pi) &= \sum_{A \in H} \mathcal{H}^f(\pi|\pi_A)\end{aligned}$$

for every partition π of the set of rows R of T. Clearly, $\sum_{A \in H} d^f(\pi_A, \pi) = \mathcal{K}^f(\pi) + \mathcal{L}^f(\pi)$.

The following inequalities hold:

$$\frac{1}{2}\sum_{A \in H} \mathcal{H}^f(\pi_A) - \frac{1}{2}n\mathcal{H}^f(\pi) \leq \mathcal{K}^f(\pi) \leq \sum_{A \in H} \mathcal{H}^f(\pi_A) \qquad (2)$$

and

$$\mathcal{L}^f(\pi) \leq n\mathcal{H}^f(\pi). \qquad (3)$$

Starting from $\mathcal{K}^f$ and $\mathcal{L}^f$ we can define fitness measures for a chromosomial population $\{K_1, \ldots, K_n\}$ as:

$$\begin{aligned}\text{fitness}^f_{\mathcal{K}}(K) &= \frac{\max\{\mathcal{K}^f(\pi_{K_i})|1 \leq i \leq n\}}{\mathcal{K}^f(\pi_K)}\\ \text{fitness}^f_{\mathcal{L}}(K) &= \frac{\max\{\mathcal{L}^f(\pi_{K_i})|1 \leq i \leq n\}}{\mathcal{L}^f(\pi_K)}\\ \text{fitness}^f_{\mathcal{K}+\mathcal{L}}(K) &= \frac{\max\{\mathcal{K}^f(\pi_{K_i}) + \mathcal{L}^f(\pi_{K_i})|1 \leq i \leq n\}}{\mathcal{K}^f(\pi_K) + \mathcal{L}^f(\pi_K)}\end{aligned}$$

for every chromosome K whose associated partition is π_K.

Note that if a chromosome has a small $\mathcal{K}^f(\pi_{K_i})$ value then its chromosomial fitness $\text{fitness}^f_{\mathcal{K}}(K_i)$ will be large. This property holds also for $\mathcal{L}^f(\pi_{K_i})$ versus $\text{fitness}^f_{\mathcal{L}}(K_i)$, and for $\mathcal{K}^f(\pi_{K_i}) + \mathcal{L}^f(\pi_{K_i})$ versus $\text{fitness}^f_{\mathcal{K}+\mathcal{L}}(K_i)$.

In Subsection 4.2 we will discuss the uses of each of these fitness measures and their impact on the final clustering. In what follows we will only focus on using $\text{fitness}^f_{\mathcal{K}}(K)$.

Our genetic algorithm has the following input parameters:

- M, the cardinality of the chromosomial population;
- N, the number of rows in the table T;
- n, the number of columns in the table T;
- k, the number of classes in the median partition;

- r, the percent of chromosomes that will be used for crossover;
- m, the percent of chromosomes that will be subject to a mutation;
- N_{max}, the number of consecutive iterations without improvement after which the algorithm will stop;
- ϵ, the margin error for the fitness function, and
- the impurity measure f.

If in the chromosomial population there exists a chromosome K_i with sufficiently large fitness, that is, a chromosome such that $\mathcal{K}^f(\pi_{K_i}) < \epsilon$, then, the algorithm stops and produces the approximation of the median partition generated by this chromosome. The impurity measure f can be the Gini index or the Shannon entropy or any valid generator function.

The number of chromosomes that will be crossed over is given by $\max\{2, rM\}$. The number of chromosomes that will be mutated is given by $\max\{1, mM\}$. The number of chromosomes that will be copied directly in the new population is given by $(1 - r - m)M$, such that at most $M - 3$ chromosomes are copied in the new generation.

The initial population of chromosomes is filled randomly with values between 1 and k. For each chromosome K_i we compute the value $\mathcal{K}^f(\pi_{K_i})$. In each population exists a chromosome which has the smallest $\mathcal{K}^f(\pi_{K_i})$ value (and so, the highest value of the chromosomial fitness $\text{fitness}^f_{\mathcal{K}}(K_i)$). We will denote this chromosome with K_{best}, its fitness with $\text{fitness}_{\text{best}}$.

If we have $\mathcal{K}^f(\pi_{K_{\text{best}}}) < \epsilon$, then the algorithm will stop, and the almost-median partition will be the one determined by K_{best}.

We will indicate the iteration number as a superscript; in other words, K_i^l is the chromosome K_i at the step l in the execution of the program and fitness_i^l would mean the value of the fitness associated with chromosome K_i at step l in the execution. If in N_{max} consecutive iterations there is no improvement in the best fitness, then the algorithm will halt and the almost-median partition will correspond to the chromosome with the best fitness at iteration N_{max}: $K_{\text{best}}^{N_{max}}$.

If the above two exit conditions do not hold, a new population is computed in the following way. The first $(1 - r - m)M$ chromosomes having the best fitness values are copied directly in the new population. The selection of these chromosomes is done by sorting the chromosomes in descending order of their fitness values and choosing the first $(1-r-m)M$ chromosomes from the sorted list. This is based on the observation that for a greater value of the fitness, the partition associated with the chromosome is closer to the median partition. This selection ensures that we will never lose the best chromosome from the old population. This strategy in which the fittest chromosomes are copied straight from the old generation to the new generation is known in the literature as elite selection [Mic99]. Next, a number of $\max\{2, rM\}$ chromosomes from the old generation are selected probabilistically to be used in generating new offsprings by crossover and the newly generated offsprings are copied into the new population. Finally, a number of $\max\{1, mM\}$ chromosomes are selected with uniform probability and subjected to mutation.

The pseudocode of the algorithm is the following:

```
initialize the population of genetic algorithm
while (true)
     compute the fitness of chromosomes in the population;
     if (fitness^f_K(K_best) < ε or there have been no relative
        improvement in best fitness value for N_max iterations)
        then
        {
        output the partition of K_best;
        exit;
        }
     copy fittest (1 − r − m)M chromosomes to new population;
     select probabilistically max{2, rM} chromosomes to cross over;
     apply crossover operator to the selected chromosomes
        and copy the offsprings to the new population;
     select with uniform probability max{1, mM} chromosomes to mutate;
     apply mutation operator to the selected chromosomes
        and copy the modified chromosomes to the new population;
     replace old population with the new one;
```

This algorithm can also use the other two fitness measures.

3.1 Crosssover and mutation methods

Some of the crossover methods that we implemented are based on manipulating the classes of the partition determined by a chromosome. We introduce several measures that reflect how good a class from the partition of the chromosome is with respect to the attribute partitions of the database.

Let $\pi_K = \{C_1, C_2, \ldots, C_k\}$ be the partition of chromosome K. We consider several fitness measures for the classes of π_K:

- The measure cf_{prod} is given by $\mathsf{cf}_{prod}(C_i) = \frac{1}{|C_i| \times \mathsf{imp}^f_T(C_i)}$, where
$$\mathsf{imp}^f_T(C_i) = \sum_{A \in H} \mathsf{imp}^f_{\pi_A}(C_i),$$
- The measure cf_{pow} is defined by $\mathsf{cf}_{pow}(C_i) = \frac{|C_i|}{2^{\mathsf{imp}^f_T(C_i)}}$, where $\mathsf{imp}^f_T(C_i)$ is defined as above.
- The symmetric difference measure cf_{sd} is
$$\mathsf{cf}_{sd}(C_i) = \left(avg_{A \in H} \left(min_{j=1}^{a_A} \frac{|C_i| + |B^A_j| - 2 \times |B^A_j \cap C_i|}{|C_i| + |B^A_j|} \right) \right)^{-1},$$

where $\pi_A = \{B_1^A, B_2^A, \dots B_{a_A}^A\}$ and n is the number of attributes in the database.

We implemented three types of crossover methods.

- (`CROSS-RAND`) This is the classical single point crossover. Starting from two chromosomes K_i and K_j, a random crossing point (called a *crossover site* is selected as a number l between 1 and N (the number of rows in the table T). The offsprings will contain the first 1 to l positions from the first parent and the last $l+1$ to N positions from the second parent and vice versa.

  For example, if $N = 6$ and $K_i = [1\ 1\ 2\ 1\ 2\ 2]$ and $K_j = [1\ 2\ 2\ 1\ 2\ 1]$ and $l = 3$ the offsprings will be $[1\ 1\ 2\ 1\ 2\ 1]$ and $[1\ 2\ 2\ 1\ 2\ 2]$.

  This representation of chromosomes has the disadvantage that two different encodings can represent in fact the same partition; this may result in redundancy in the chromosomial population. For example, the partitions $K_1 = [1\ 1\ 1\ 2\ 3\ 2]$ and $K_2 = [2\ 2\ 2\ 3\ 1\ 3]$ are identical.

  Also, the number of classes of the chromosomes can change during crossover. For example, if we cross over K_1 and K_2 at the crossover site 3 the offsprings will be $[1\ 1\ 1\ 3\ 1\ 3]$ and $[2\ 2\ 2\ 2\ 3\ 2]$. Thus, from chromosomes representing partitions with three classes we have generated chromosomes representing partitions that have only two classes. The number of classes can also increase, for example if we reverse the above crossover operation, however it will never be higher than the targeted number of classes since no new class identifiers are introduced.

- (`CROSS-BEST-CLASS`) To generate unique encodings of a partition we will translate the vector representation into a representation using an ordered list of ordered lists. For example, if $K_1 = [1\ 1\ 1\ 2\ 2\ 2]$ and $K_2 = [2\ 2\ 2\ 1\ 1\ 1]$, then K_1 will be represented as $[[1\ 2\ 3][4\ 5\ 6]]$, meaning that the partition has two classes, the first class contains the rows 1, 2 and 3 and the second class contains the rows 4, 5 and 6 from the table. The list of lists is ordered with respect to the smallest row number of each list which are themselves ordered by the row numbers. This way the representation of the partition corresponding to K_2 is also $[[1\ 2\ 3][4\ 5\ 6]]$.

  For a chromosome K_i we will denote the representation that uses an ordered list of ordered lists by OL_i.

  This crossover operator uses the list of lists representation. The `CROSS-BEST-CLASS` method selects the best class in each chromosome with respect to a specific class fitness measure. Then, the best classes are exchanged between the two parent chromosomes.

  The crossover of two chromosomes K_i and K_j is performed in the following way:

  - Get the list of lists representations, OL_i and OL_j, for the two chromosomes.

  - Choose the best classes that will be exchanged between the two chromosomes: class c_i from OL_i and class c_j from OL_j.
  - Copy class c_j instead of class c_i into OL_i. Find the elements of class c_j that appear in other classes of OL_i and remove them from those classes.

    If this operation has not emptied any classes in OL_i and if there are elements in c_i that do not appear in c_j, redistribute them probabilistically to the other classes of OL_i (classes with smaller sizes will have greater chances to receive the new elements).

    Otherwise, if some classes in OL_i became empty, and there are elements to redistribute from $c_i - c_j$, then redistribute these elements to the empty classes. If there are more elements from $c_i - c_j$ to redistribute than we have empty classes, then redistribute these elements with equally among the empty classes. If there are no elements to redistribute, take some elements from other classes (different from class c_i) and move them to the empty classes. The classes from where an element will be taken are picked randomly.
  - A similar technique is used for replacing c_j in OL_j by c_i.

  Finally, the classes of chromosome K_i are set according to the grouping in OL_i and the classes of chromosome K_j are set according to the grouping in OL_j.

- (`CROSS-BEST-WORST`) This crossover method also uses the list of lists representations of the two chromosomes subject to crossover. Two classes are selected for each chromosome, the best class and the worst class with respect to a certain class fitness measure. Then, the worst class in one partition is replaced with the best class in the other partition and vice versa.

  In this process some elements from the worst class need to be redistributed to other classes; this redistribution can be done probabilistically such that classes with smaller sizes have greater chances to receive the new elements.

We implemented two types of mutation operators.

- (`MUT-RAND`) The first type of mutation involves changing randomly a number of $\max\{1, 0.1N\}$ positions in the chromosomes. The new value for each chromosome position is chosen randomly from $1 \dots k$.

- (`MUT-MOVE`) This type of mutation operator uses the idea of the list of lists representation and involves moving a number of $\max\{1, 0.1N\}$ elements to randomly chosen classes of the selected chromosomes.

  For a selected chromosome K_i the list of lists representation OL_i is generated first. Then, between two randomly selected classes of this chromosome, one randomly selected element from the first class will be moved to the second class. After the exchange takes place the partition associated with K_i is modified to reflect the changes done in OL_i.

4 Experimental Results and Robustness of the Proposed Algorithms

To test our program we generated a synthetic database with N rows, n columns, k distinct values in the domain of the attributes and a reference partition π_{ref}.

All attributes of the table T involved in the experiment are numeric and they have the same domain containing k distinct numbers between 1 and k. To generate each row of T we selected a random number between 1 and k that was placed in the reference partition on the position associated with the row. We saved this number in all columns of T with the exception of one randomly chosen column, into which we saved another random value between 1 and k. This database generation procedure ensures that in the reference partition we have an almost-median partition. The integer values that are stored in the reference partition π_{ref} and in the rows of the database are generated with uniform probability. Thus the attribute partitions and the reference partition π_{ref} have classes with similar cardinalities.

To conclude how well the genetic algorithms perform we computed the distance $d^f(\pi_{\text{ref}}, \pi*)$ between the partition $\pi*$ generated by the algorithms and the reference partition π_{ref}. In all these experiments we have used $\text{fitness}^f_{\mathcal{K}}(K)$ as a measure of the fitness of a chromosomial population.

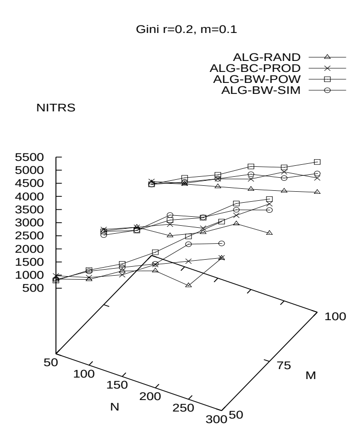
The algorithms that we examine are described in the following table:

Algorithm Notation	Crossover Method	Mutation Method	Class Fitness Measure
ALG-RAND	CROSS-RAND	MUT-RAND	N/A
ALG-BC-PROD	CROSS-BEST-CLASS	MUT-MOVE	cf_{prod}
ALG-BW-POW	CROSS-BEST-WORST	MUT-MOVE	cf_{pow}
ALG-BW-SIM	CROSS-BEST-WORST	MUT-MOVE	cf_{sd}

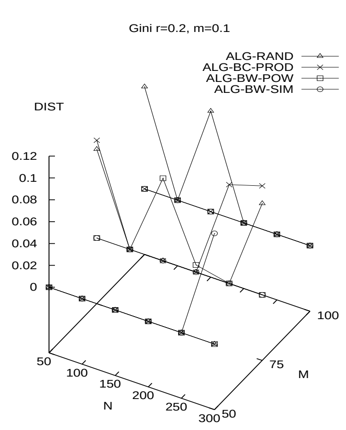
To evaluate the dependency of the performance of the proposed algorithms on the size of the input table we tested the above four genetic algorithms, using a fixed number of chromosomes on databases with various numbers of rows and a fixed number of columns.

We tested the algorithms for three different sizes of the chromosomial population of 50, 75 and 100 chromosomes and, in each case, we experimented with 50, 100, 150, 200, 250 and 300 rows. All the other parameters of the genetic algorithms were kept constant across all the runs; namely, we worked with $k = 5$, a crossover rate $r = 0.2$, and a mutation rate $m = 0.1$. The number of consecutive iterations without improvement was set as $N_{max} = 500$, the error margin for the dissimilarity of the chromosomes in the population with respect to the database was $\epsilon = 0.00001$, and we used the Gini index in all cases.

The results for the four algorithms are shown in Figure 1. Each line in this picture represents the results obtained for a specific value of the number of chromosomes M and a specific algorithm.



Number of iterations versus M and N



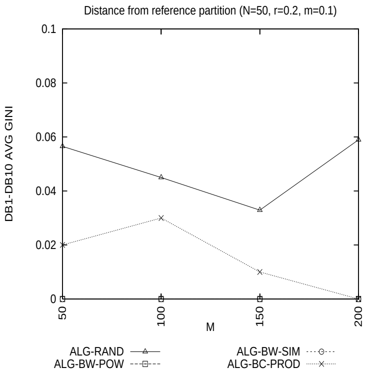
Distance from reference partition versus M and N

Figure 1: Results of the first experiment

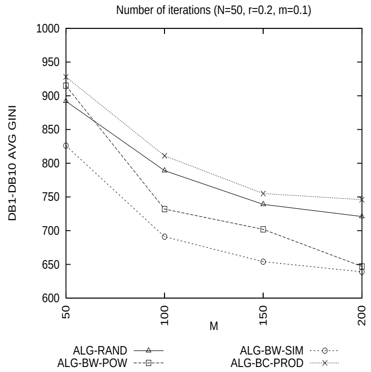
From the first graphic of Figure 1, we can observe that `ALG-RAND` converges in a fewer number of iterations than the other algorithms. From the second graphic we can notice that all algorithms found the reference partition or a close approximation of it. The best results were obtained by `ALG-BW-SIM` and `ALG-BW-POW` which, except for one or two cases, always found the reference partition.

A second experiment consisted of running the genetic algorithms on 10 different databases with a fixed number of rows and columns. For each database we have run the algorithms with a number of chromosomes equal to 1, 2, 3, and 4 times the number of rows in the database. The goal of this experiment was to see how the algorithms behave on average over all databases and to get an idea about how they behave when the population of chromosomes increases relative to the size of the database. For each combination of the values of N and M the algorithms have been executed three times, each execution using a different seed value for the random number generator (thus leading to different initializations of the chromosomial population). The results are shown in Figure 2 and they represent an average over all executions on all databases.

Looking at these results we can observe that again, `ALG-BW-SIM` and `ALG-BW-POW` have performed extremely well and this time found the reference partition in all cases. The other two algorithms, `ALG-RAND` and `ALG-BC-PROD`, also performed well and converged to close approximations of the reference partition. In terms of number of iterations the best algorithm was `ALG-BW-SIM`, followed by `ALG-BW-POW`, `ALG-RAND` and `ALG-BC-PROD`. `ALG-RAND` performed worse in this experiment with respect to the number of iterations since it did not always converge to the reference partition and thus its search took longer. Another important observation is that by increasing the size of the chromosomial



Average distance from reference partition



Average number of iterations

Figure 2: Results of the second experiment involving 10 databases

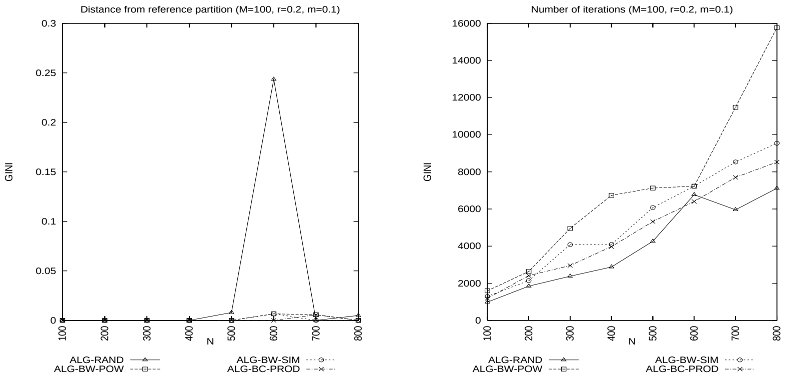
population the number of iterations required by the algorithms decreases significantly.

A third experiment consisted of running the algorithms on databases with increasing sizes $N \in \{100, 200, 300, 400, 500, 600, 700, 800\}$, a fixed number of chromosomes $M = 100$, $n = 5$ columns, and $d = 5$ distinct values in the domain of the attributes. The impurity measures used were the Gini index and the Shannon entropy.

In the experiments that used the Gini measure for the conditional impurity (see Figure 3) all of the algorithms converged to the reference partition or to a close approximation of it. `ALG-BW-SIM`, `ALG-BC-PROD`, `ALG-BW-POW` performed very well and in the few cases when they did not converge to the reference partition they have found very close approximations. `ALG-RAND` also performed pretty well, with the exception of the experiment for $N = 600$, when it has converged to a partition at distance 0.24349 from the reference partition. For this experiment we can also notice the increase of the number of iterations required by `ALG-RAND`. Regarding the number of iterations, with the exception of `ALG-BW-POW`, the algorithms exhibit a linear dependence on the size of the database; however `ALG-BW-POW`'s performance is non-linear.

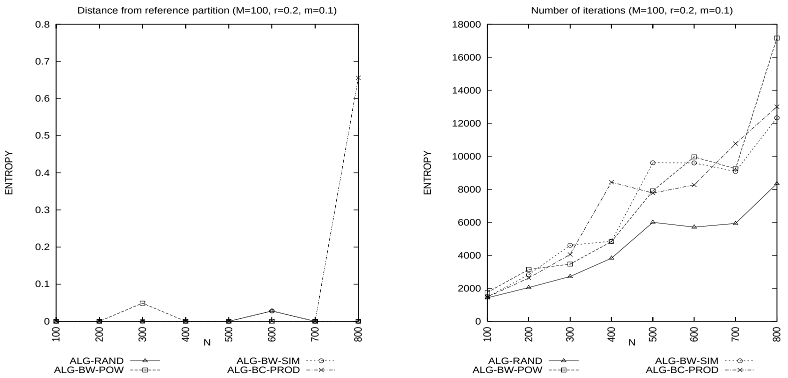
Replacing the Gini measure by the Shannon entropy measure yields the results shown in Figure 4.

For the experiment using the entropy measure, we note that the algorithms find the reference partition in all but one case. `ALG-BC-PROD` failed to converge to a partition close to the reference partition for the case when $N = 800$, while the other algorithms found closer approximations when they could not converge toward the reference partition. `ALG-BW-POW`'s performance regarding the number of iterations is not so good, and based also on the results of the previous experiment we can conclude that the number of iterations required by this algorithm is rather impredictable. Another conclusion of these experiments



Distance to reference partition   Number of iterations

Figure 3: Distance to reference partition and number of iterations when using the Gini measure



Distance to reference partition   Number of iterations

Figure 4: Distance to reference partition and number of iterations when using the Shannon entropy measure

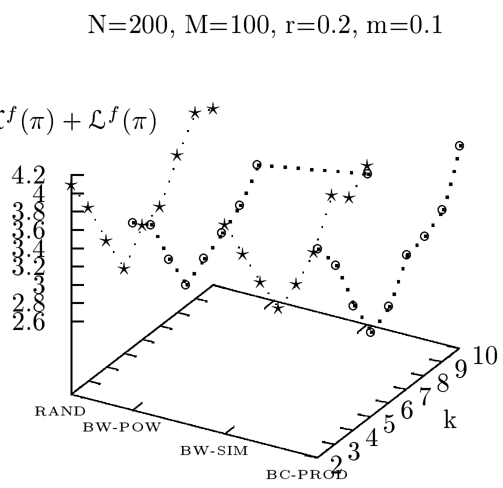


Figure 5: Determining the number of natural classes using the Gini measure with different values of k

is that using the Gini measure makes the algorithms converge in fewer iterations than when using the entropy.

4.1 Determining the number of natural classes

For real-life databases it is seldom the case that the number of clusters k is known apriori. Therefore, to demonstrate the robustness of our approach we probe the synthetic database (whose number of "natural clusters" is known) by seeking clusterings π_K for successive k values, $k = 2, 3, \ldots$ and computing the dissimilarity $\sum_{i=1}^{n} d^f(\pi_{A_i}, \pi_K)$. The experimental results show that the dissimilarity reaches a minimum for the value of k that corresponds to the "hidden" clustering, as shown in Figures 5, 6, 7.

The results shown in Figures 5, 6, 7 were obtained by running the four genetic algorithms using three different generator functions $f \in \{Gini,\ entropy,\ peak\}$ on one database with $N = 200$ rows, $M = 100$ chromosomes, $n = 5$ columns, and $d = 5$ distinct values in the domain of the attributes. The crossover rate was set to $r = 0.2$ and the mutation rate to $m = 0.1$. The number of classes in the almost-median partition was specified as $k \in \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

Each plot presents the evolution of the dissimilarity $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$ as a function of k, where π^* is the almost-median partition produced by the genetic algorithm. We can observe that for all algorithms and for all generators f the dissimilarity $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$ has a minimum value when $k = 5$ which in our

N=200, M=100, r=0.2, m=0.1

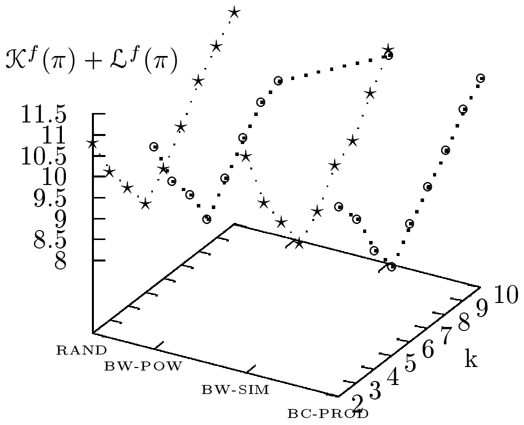


Figure 6: Determining the number of natural classes using the Shannon entropy measure with different values of k

N=200, M=100, r=0.2, m=0.1

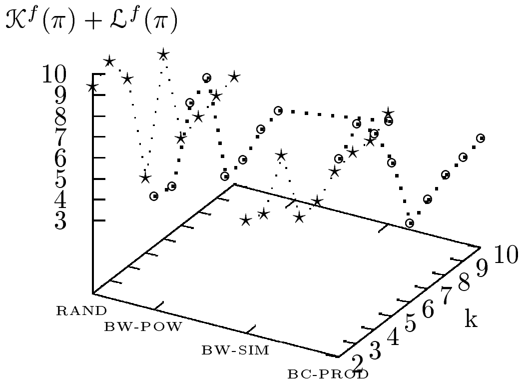


Figure 7: Determining the number of natural classes using the peak measure with different values of k

experiment represents the real number of clusters embedded in the data. This is an important result since it provides us with a method for determining the number of clusters embedded in data in the absence of any apriori knowledge regarding the clustering. This method would require that we find the value k that minimizes $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$. This value would represent the number of natural clusters existing in the data.

4.2 Analysis of the different fitness measures

If no constraint is imposed on the number of classes of the partition sought by a genetic algorithm, then by using the fitness measure $\text{fitness}^f_{\mathcal{K}}$ the algorithm will evolve toward a partition $\bigcap_{A \in H} \pi_A$ whose classes fit inside the classes of the attribute partitions. On the other hand, if the fitness measure $\text{fitness}^f_{\mathcal{L}}$ will be used, then the genetic algorithm will evolve toward the one-class partition ω_R. Note that the inequalities 2 impose an upper bound on $\mathcal{K}^f(\pi)$ that depends on the database, and a lower bound that depends on both the database and on the entropy of the partition π obtained by the genetic algorithm. If the algorithm maximizes the fitness $\text{fitness}^f_{\mathcal{K}}$, the tendency of the algorithm will be to evolve toward partitions that have small values for $\mathcal{K}^f(\pi)$, and, therefore, large values of $\mathcal{H}^f(\pi)$. This would imply that the majority of the classes of π will have approximatively the same cardinalities. Conversely, the fitness measure $\text{fitness}^f_{\mathcal{L}}(K)$ is large when $\mathcal{L}^f(K)$ is small; this happens when π has a few classes significantly larger than the others, as follows from inequality 3.

Suppose that we are seeking a partition $\pi*$ with k classes. In this case, the largest value of the entropy $\mathcal{H}^f(\pi*)$ is $k \cdot f\left(\frac{1}{k}\right)$, so

$$k \cdot f\left(\frac{1}{k}\right) \geq \mathcal{H}^f(\pi*) \geq \frac{n}{2}\left(\frac{1}{2}\sum_{A \in H} \mathcal{H}^f(\pi_A) - \mathcal{K}^f(\pi*)\right),$$

which implies

$$\mathcal{K}^f(\pi*) \geq \frac{1}{2}\sum_{A \in H} \mathcal{H}^f(\pi_A) - \frac{nk}{2} \cdot f\left(\frac{1}{k}\right).$$

Note that for every generator function f the quantity $k \cdot f\left(\frac{1}{k}\right)$ increases with k, so larger values of k allow us to lower the value of $\mathcal{K}^f(\pi*)$ and thus increase the value of the fitness function $\text{fitness}^f_{\mathcal{K}}$.

The choice of the fitness measure depends on the characteristics of the database. If the given database is characterized by attribute partitions having classes with similar cardinalities, then by using only the fitness measure $\text{fitness}^f_{\mathcal{K}}$, the genetic algorithm will naturally evolve toward a partition with approximatively equal sized classes. If the database is characterized by attribute partitions which have some classes significantly larger than the others, then by using $\text{fitness}^f_{\mathcal{L}}$ together with $\text{fitness}^f_{\mathcal{K}}$ (so by using $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$) in the fitness measure, the genetic algorithm will be helped in evolving toward a partition that is closer to the attribute partitions of the database and also has some larger classes.

ALG-BW-SIM			
	$d_f(\pi_{\text{ref}}, \pi)$		
Fitness measure	$\mathcal{K}^f(\pi)$	$\mathcal{L}^f(\pi)$	$\mathcal{K}^f(\pi) + \mathcal{L}^f(\pi)$
No. of rows			
50	0	0.498333	0.225474
100	0	0.592425	0.186397
150	0	0.8225	0.372595
200	0.277767	0.816678	0.411282

ALG-RAND			
	$d_f(\pi_{\text{ref}}, \pi)$		
Fitness measure	$\mathcal{K}^f(\pi)$	$\mathcal{L}^f(\pi)$	$\mathcal{K}^f(\pi) + \mathcal{L}^f(\pi)$
No. of rows			
50	0	0.657273	0.226087
100	0	0.398033	0
150	0	0.578803	0
200	0	0.38889	0.168676

Figure 8: Results for databases that have moderate classes

Experiments done with the random version of the genetic algorithm `ALG-RAND` and the genetic algorithm `ALG-BW-SIM` show that the measure $\mathcal{K}^f(\pi*)$ performs well for the case of a database with moderate classes (the measure $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$ performs well too, but $\mathcal{K}^f(\pi*)$ performs better) and in the case of a database with one large class in the attribute partitions, the measure $\mathcal{K}^f(\pi*)$ does not perform well while the measure $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$ does indeed perform better.

In the case of the algorithm `ALG-BW-SIM` the $\mathcal{L}^f(\pi*)$ measure has a strong influence on the convergence process, in the sense that when $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$ is used as the fitness measure, the genetic algorithm can evolve quickly toward a partition that has a large class (therefore minimizing $\mathcal{L}^f(\pi*)$) and then is not able to escape this local minima so that it can minimize $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$.

The results for the database with moderate classes are shown in Figure 8 and the results for the database with one larger class are presented in Figure 9.

The generated databases used to test the fitness measures had $n = 5$ columns and $d = 5$ classes in the attribute partitions. The above results were obtained using the following parameters for the genetic algorithms: $M = 50$ chromosomes in the population, $k = 5$ classes in the median partition, a crossover rate $r = 0.2$, and a mutation rate $m = 0.1$. The number of consecutive iterations without improvement was set as $N_{max} = 500$, the error margin for the dissimilarity of the chromosomes in the population with respect to the database was $\epsilon = 0.00001$, and we used the Gini index in all cases.

We can observe that when `ALG-BW-SIM` is run on a database with one large class, for the case when the number of rows increases to 150 and 200, the mea-

ALG-BW-SIM			
	$d_f(\pi_{\text{ref}}, \pi*)$		
Fitness measure	$\mathcal{K}^f(\pi*)$	$\mathcal{L}^f(\pi*)$	$\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$
No. of rows			
50	0.334276	0.52053	0
100	0.363151	0.686753	0.0380912
150	0.279182	0.664308	0.352585
200	0.271919	0.655502	0.434664

ALG-RAND			
	$d_f(\pi_{\text{ref}}, \pi)$		
Fitness measure	$\mathcal{K}^f(\pi)$	$\mathcal{L}^f(\pi)$	$\mathcal{K}^f(\pi) + \mathcal{L}^f(\pi)$
No. of rows			
50	0.375758	0.460877	0.222353
100	0.358622	0.6624	0.182188
150	0.574739	0.648622	0.226466
200	0.549406	0.645865	0.112

Figure 9: Results for databases that have one large class

sure $\mathcal{K}^f(\pi*)$ produces better results than the measure $\mathcal{K}^f(\pi*) + \mathcal{L}^f(\pi*)$. When the search space becomes too large and the number of chromosomes is not sufficiently large to cope with this complexity, the effect of $\mathcal{L}^f(\pi*)$ is a negative one; the partitions with one large class predominate in the chromosomial population and prevent the generation of new partitions with more balanced classes.

The results show that in the case of moderate classes, it is best to use $\text{fitness}^f_{\mathcal{K}}(K)$. In the case when the attribute partitions of a database have a larger class, we can notice that `ALG-RAND` performs better by using $\text{fitness}^f_{\mathcal{K}+\mathcal{L}}(K)$, while `ALG-BW-SIM`'s performance depends on both the fitness measure that is used and on the number of rows of the database, which suggests the approach of running the algorithm twice using both measures and retaining the best approximation of the median partition.

5 Conclusions

In this paper we showed how the notion of generalized entropy can be used to design a number of genetic algorithms for finding clusters in data. We have experimented with different fitness measures for the evaluation of the chromosomial population and with different crossover and mutation operators. Our experiments showed that, with very few exceptions, the algorithms converged to the clustering that we embedded in the synthetically generated databases, and when they didn't converge, then they found a clustering that was close to the "hidden" one. The number of iterations required by our algorithms scales linearly with the size of the database (with the exception of the algorithm `ALG-BW-POW`).

We also showed that, using the dissimilarity measure $\mathcal{K}^f(\pi) + \mathcal{L}^f(\pi)$, we have a way of determining the number of clusters embedded in the data in the absence of any apriori knowledge.

References

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jorg Sander. Optics: Ordering points to identify the clustering structure. *Proceedings of ACM SIGMOD'99 Int. Conf. on Management of Data, Philadelphia PA*, 1999.

[AGGR98] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering for high dimensional data for data mining applications. In *Proceedings of the 1998 ACM International Conference on Management of Data*, pages 94–105, 1998.

[BL95] Jean-Pierre Barthélemy and Bruno Leclerc. The median procedure for partitions. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, 1995.

[ChWcY99] Cheng Chun-hung, Fu Ada Wai-chee, and Zhang Yi. Entropy-based subspace clustering for mining numerical data. *ACM SIGKDD International Conference on Knowledge Discovery d Data Mining, San Diego, August 15-18*, pages 84–93, 1999.

[Fal99] Emanuel Falkenauer. *Genetic algorithms and grouping problems.* Jon Wiley, New York, 1999.

[HGV97] Han Eui-Hong (Sam) Han, Karypis George, and Kumar Vipin. Clustering based on association rule hypergraph. *Proc. Workshop on Research Issues on Data Mining and Knowledge Discovery*, 1997.

[JB91] Donald R. Jones and Mark A. Beltramo. Solving partitioning problems with genetic algorithms. *Proceedings of the fourth International Conference on Genetic Algor hms*, pages 442–449, 1991.

[JMF99] A.K. Jian, M.N. Murty, and P.J. Flynn. Data clustering: A review. *ACM Computing Surveys, vol 31, no. 3*, September 1999.

[KR90] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data*. John Wiley, New York, 1990.

[MHPJX96] Ester Martin, Kriegel Hans-Peter, Sander Jorg, and Xu Xiaowei. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of 2nd Innt'l nference on Knowledge Discovery and Data Mining, Portland, Oregon*, August 1996.

[Mic99] Zbigniew Michalewicz. *Genetic algorithms + Data Structures = Evolution Programs*. Springer, New York, 1999.

[Mir96] Boris Mirkin. *Mathematical Classification and Clustering*. Kluwer Academic Publishers, Netherlands, 1996.

[Rég83a] Régnier. Sur quelques aspects mathématiques des problèmes de classification automatique. *Math. Sci. Hum., no 82*, pages 13–29, 1983.

[Rég83b] Simon Régnier. Etudes sur le polyèdre des partitions. *Math. Sci. Hum., no 82*, pages 85–111, 1983.

[SCC00a] Dan Simovici, Dana Cristofor, and Laurentiu Cristofor. Generalized entropy and projection clustering of categorical data. *Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 619–625, 2000.

[SCC00b] Dan Simovici, Dana Cristofor, and Laurentiu Cristofor. Generalized entropy and projection clustering of categorical data. Technical Report 00-3, University of Massachusetts, Boston, Massachusetts, 2000.

[SCC00c] Dan Simovici, Dana Cristofor, and Laurentiu Cristofor. Mining for purity dependencies in databases. Technical Report 00-2, University of Massachusetts, Boston, Massachusetts, 2000.

[SRK99] Guha Sudipto, Rastogi Rajeev, and Shim Kyuseok. Rock: A robust clustering algorithm for categorical attributes. *Proceedings of the IEEE International Conference on Data Engineering, Sydney, March 1999*, March 1999.

[TRL96] Zhang Tian, Raghu Ramakrishnan, and Miron Livny. Birch: A new data clustering algorithm and its applications. *Proceedings of the ACM SIGMOD Conference on Management of Data, ntreal, Canada*, June 1996.