CHAPTER 3

The Fuzzy Guided Genetic Algorithm for Data Clustering

3.1 In a Nutshell

This chapter presents a detailed discussion of the proposed algorithm for the clustering of sequential data based on multiple criteria using a soft computing approach. The major computational elements of the algorithm and its salient features are explained. The algorithm will always generate a set of clusters but the quality of these clusters has to be proven. A cluster quality index has been formulated for this purpose. As for any algorithm, the time and space complexity issues have been studied. Lastly, a comparison with existing algorithms is made before concluding the chapter.

3.2 The Problem

In Chapter 2, the two classes of traditional clustering algorithms, namely, hierarchical and partitional schemes were discussed. There are also statistical, fuzzy, neural, evolutionary and knowledge-based approaches to clustering.

The clustering algorithm developed in this work is applicable to sequential data. Sequential data, here, is defined as a data stream that is naturally ordered according to some ordering criterion like time in a time series. The basic assumption is that when such data is clustered into groups, the clusters form contiguous blocks. Sequential data clustering can be considered as a special case of the general data clustering problem. It can also be defined as data segmentation or data grouping.
The data items (or data objects) in the data set have to be grouped into clusters given the fact that there are one or more criteria that are influential in their formation. These criteria can be qualitative as well as quantitative. The interplay of the criteria, their relative importance and our level of confidence in them may vary.

The problem can be decomposed into two sub-problems. Firstly, there has to be a method to arrive at a combined score for the fitness of a set of data items that have formed a cluster. Secondly, there should be a way for changing the constitution of the clusters so as to maximize the fitness scores of clusters. The problem can be thought of as a multi-objective optimization problem. In the singular case, if the most important or predominant criterion can be identified, then it reduces to a single objective optimization problem, namely optimization \( w.r.t. \) the predominant criterion. Hence, the problem can be stated as follows.

**Problem Definition:** Partitioning a given data set that is ordered on the basis of some criterion into clusters such that the clusters are non-overlapping, contiguous and optimal \( w.r.t. \) one or more criteria.

Let a data set \( X \) consist of \( N \) data items.

\[
X = \{ x_1, x_2, \ldots, x_N \} \quad \text{where } x_i \text{ is the } i^{th} \text{ data item.}
\]

A partition \( C \) of \( X \) breaks \( X \) into subsets \( \{ C_1, C_2, \ldots, C_m \} \) satisfying the following conditions:

\[
C_i \cap C_j = \emptyset \quad \text{for all } i \text{ and } j \text{ varying from 1 to } m, \ i \neq j
\]

\[
C_1 \cup C_2 \cup \ldots \cup C_m = X
\]

\[
C_i = \{ x_{i1}, x_{i1+1}, \ldots, x_{i1+k-1} \} \quad \text{where } C_i \text{ is the } i^{th} \text{ partition and } x_{i1}, x_{i1+1}, \ldots, x_{i1+k-1} \text{ are consecutive data items in } X.
\]

### 3.3 The Model Overview

The model uses a soft computing approach to find clusters in the data. The two pillars of soft computing, namely, genetic algorithms (GA) and fuzzy logic have been combined to yield a multi-objective optimization technique. The GA starts with an initial population, where each individual of the population is a different partitioning of the data set.
generated by different threshold values of the ordering criterion. A fuzzy fitness finding mechanism guides the GA through the search space by combining the contributions of various criteria/features that have been identified as the governing factors for the formation of the clusters. Each feature is modeled as a fuzzy variable and intuitive fuzzy rules combine the fuzzy inputs to yield a single fitness value (in %) for every cluster. The individuals of the population are “evolved” through progressively fitter generations of clustering arrangements obtained by the application of genetic operators. The GA terminates when there is no significant improvement in the fitness values of clusters. The model uses a fuzzy logic based fitness finding mechanism to guide the GA through its search space. A schematic diagram of the model is given in Fig 3.1.

![Fig. 3.1: Model of the Fuzzy Guided Genetic Algorithm (FGA)](image)

### 3.4 The Algorithm

The algorithm has two computational elements that work together.

i) The Genetic Algorithm (GA) and

ii) The Fuzzy Fitness Finder (FFF).

The main algorithm is the GA and the FFF can be thought of, as embedded within the GA. For clarity, the two algorithms will be dealt with separately. The flowchart of the integrated algorithm is given in Fig. 3.2 and Fig. 3.3. The pseudocode is given by Algorithm 3.1.
Create Initial Population (size n)
Individual = \{cluster1, cluster2...\}

∀ individual I = 1,n
\[ F_I = \sum_{i=1}^{m} \text{fitness}_i \]
m clusters in I

Selection

Crossover and Mutation

Fitness Calculation

Terminating Condition

\( \alpha \)

\( n \)

Fig. 3.2: Flowchart of Fuzzy Guided Genetic Algorithm (GA Part)
Algorithm (FGA)

begin
  initialize    // data and variables
    populationsize,
    population = { x_i }, i = 1, ..., populationsize
    x_i = set of clusters in the i^{th} individual
  while (not termination-condition) do
    begin
      call GA (1 iteration) which makes calls to FFF
      population = newpopulation = { y_i }
      y_i = new clusters in the i^{th} individual
    end
  return population    // population of optimal clusters
end

Algorithm 3.1: Fuzzy Guided Genetic Algorithm (FGA)
3.4.1 The Genetic Algorithm Part

The basic model of a simple genetic algorithm has been implemented. The five components of a GA are:

- A genetic **representation** for potential solutions to the problem.
- Method to create an **initial population** of potential solutions
- An evaluation function to rate solutions in terms of their “**fitness**”
- **Selection** of individuals for the next generation
- **Genetic operators** that alter the composition of the children

In order to make a successful run of a GA, the values for the parameters of the GA have to be defined like the population size, parameters for the genetic operators and the terminating condition.

The implementation details of each of these components is discussed separately in this section but the fitness evaluation method (FFF) which is invoked from the GA will be dealt with in the next section.

3.4.1.1 Representation

The most common data structure used to represent individuals in a population are strings. But, depending on the problem, any suitable data structure can be used. Here, each individual of the population is a different clustering arrangement of the same data set. So, an individual can be represented as an array of integers where each integer stands for the cluster number to which the data point at that position belongs. If there are \( N \) elements in the data set and a particular initial clustering results in \( m \) clusters, then an individual, for example, could possibly look like

\[
\text{Individual } i = \{1, 1, 1, 2, 3, 3, 4, 4, \ldots, m, m, m\}
\]

where the first three data elements belong to cluster 1, the fourth belongs to cluster 2, fifth and sixth to cluster 3 etc. and the last three data elements to cluster \( m \). To generalize, the \( i^{th} \) individual of the population is coded as an integer array of size \( N \) with each element in the array taking a value in the range \((1,m_i)\), \( m_i \) being the number of
clusters in the i\textsuperscript{th} individual. In every individual, the size of the array remains the same (= N) but the value of m\textsubscript{i} changes as each partitioning could result in a different number of clusters which need not be same. Choice of an integer array makes it computationally simple to store the population and makes it easy to implement the genetic operators as can be seen further ahead in this section.

3.4.1.2 Initial Population

A population of \( n \) individuals holds \( n \) alternate solutions to the problem of partitioning the data set of size \( N \) into clusters. The individuals are created by taking different cut-off values or thresholds for the ordering criterion. The clustering starts with one cluster formed by the first data element and consecutive data items are added to the cluster until the distance to the next data element exceeds the cut-off value, in which case, a new cluster is started from that element onwards. For example, grouping according to different time intervals in a web log data series will generate new individuals. The distance in this case is the time interval between two logged data items. The pseudocode for the algorithm for the creation of the initial population is given in Algorithm 3.2.

\begin{algorithm}
\textbf{Algorithm (Create Initial Population)}
\begin{algorithmic}
\State \textbf{initialize} // data and variables
\State \textbf{initialize} \( n \) = populationsize,
\State \textbf{initialize} \( X = \{x_1, x_2, \ldots, x_N\} \), \( N \) = size of the data set
\State \textbf{initialize} \( \text{cut}_i \), \( i = 1, \ldots, n \) // \( n \) selected cut-off values
\State \textbf{initialize} \( i = 0 \)
\Do \( i = i + 1 \)
\State \textbf{initialize} \( j = 1 \)
\Do \( j = j + 1 \)
\If {distance\( (x_{j-1}, x_j) > \text{cut}_i \)} \( y_j = y_{j-1} + 1 \)
\Else \( y_j = y_{j-1} \)
\EndIf
\Until {j = N}
\State \textbf{initialize} \( \text{individual}_i = \{y_1, y_2, \ldots, y_N\} \)
\Until {i = n}
\State \textbf{return} \( \text{population} = \text{individual}_i \), \( i = 1, \ldots, n \)
\End
\end{algorithmic}

Algorithm 3.2: Creation of initial population
The choice of cut-off values (as many as the population size) is problem specific. Generally, the limits for the cut-off values come from domain knowledge. The implementer would know how far apart the data items could possibly be, if they have to be grouped together. A cluster will be characterized by the property that the distance between consecutive data items in a cluster cannot exceed a cut-off value. It can be seen that smaller cut-offs lead to larger number of small-sized clusters whereas larger cut-offs lead to smaller number of large-sized clusters.

3.4.1.3 Selection

The Selection operator (also known as the reproduction operator) decides which of the individuals in the population will go into the next generation. This is decided by the fitness value of an individual as calculated by the FFF that will be taken up in the next section. At this point the assumption is that a fitness value pertaining to each individual is available.

Individuals with a higher fitness value should have a higher probability of contributing one or more offspring to the next generation. The biased roulette wheel which is a standard selection method was used where each individual of the current population has a roulette wheel slot, sized in proportion to its fitness.

\[ Wf_i = \frac{f_i}{\sum_{i=1}^{n} f_i} \times 100 \]

where \( Wf_i \) = weighted roulette wheel value (in %) for individual i

\( n \) = population size

\( f_i \) = fitness of the i\textsuperscript{th} individual.

To select a new population, the weighted roulette wheel is spun \( n \) times. \( Individual_i \) is given \( Wf_i \) percentage of the biased wheel, so each spin can turn up \( Individual_i \) with probability \( Wf_i \). In this way, fitter individuals have a greater chance of having a higher number of offspring in the succeeding generation.

In the pseudocode for the Algorithm Select (Algorithm 3.3), the value returned is the population index value corresponding to the selected individual. Algorithm Select has to
be called \( n \) times to generate the new population. A random choice routine called \( \text{random} \) is used for Selection. It returns a real pseudorandom number between zero and one.

**Algorithm (Select)**

```plaintext
// roulette wheel selection
begin
initialize // input data and variables
n = populationsize
fitness[i], i = 1,n
sumfitness // summation of fitness[i], i =1,n
slotsum = 0
j = 0
rand = random( ) * sumfitness // wheel point
repeat // find wheel slot
    j = j + 1
    slotsum = slotsum + fitness[j]
until (slotsum >= rand)
return j // selected individual’s index number
end
```

Algorithm 3.3: Selection

The real variable \( \text{rand} \) contains the location where the wheel has stopped after a random spin. The partial sum of fitness values is cumulated in the variable \( \text{slotsum} \). The repeat-until construct searches through the weighted roulette wheel until the partial sum is greater than or equal to the stopping point \( \text{rand} \).

### 3.4.1.4 Genetic Operator: Crossover

Crossover is the most widely used recombination operator. Uniform 1-point crossover has been used. In general, 1-point crossover selects a random cut point and combines the first portion of one parent with the second portion of the other and vice versa to produce two offspring. The individual here consists of an array of cluster numbers. Hence, the main issue in recombination is the renumbering of clusters in the resulting offspring. Firstly, the steps involved in the crossover are enumerated. Then the various cases that arise in renumbering of clusters are studied.
1. The individuals formed after selection are paired or mated at random. This results in $n/2$ pairs of individuals in a population of size $n$. If $n$ is even, there will be $n$ new individuals formed in each generation, otherwise there will be only $(n-1)$ individuals. In the implementation, we have taken $n$ to be even.

2. In each pair of individuals of step 1 above, an integer position $k$ along the length of the individual (length $N$) is selected uniformly at random between 1 and the length of the individual less one $[1,N-1]$.

3. Two new individuals are created by renumbering all clusters from position $k+1$ to $N$ in both the pairing parents.

A visualization of crossover is shown in Fig. 3.4 and Algorithm 3.4 gives the pseudocode.

**Algorithm (Crossover)**

```
begin
  initialize // input data and variables
  n = populationsize
  population[i], i = 1,n  // each element is an array of integers
  selected[i], i = 1,n    // the index of the selected individuals
  i = 0
  j = 1
  do  i = i + 1
      p1 = selected[j]
      p2 = selected[j+1] // parents for crossover
      k = select_random_number(1, N-1) // to select crossover point
      renumber(population[p1], population[p2]) // renumber clusters in population[p1] & population[p2] from k+1 to N
      j = j + 2
  until i > n/2
  return new population
end
```

Algorithm 3.4: Crossover
An issue that arises in renumbering of clusters is that the crossover cut point can be anywhere within a cluster or exactly at a border of a cluster. A cut point at \( k \ (1 < k < N) \) is defined as a border if the cluster number of the data item at \( k \) is not the same as that at \( k+1 \). This implies that we are looking at only right hand side borders. There are 4 cases depending on where the crossover point \( k \) happens to fall. In a crossover between parent A (or Parent1) and parent B (or Parent2), let the cut point be at \( k \) and the cluster numbers at the \( k^{th} \) positions of A and B be \( a \) and \( b \) respectively.

**Case 1:** The cut point cuts both A and B at borders. This is the simplest case and falls in the category illustrated in Fig. 3.4. The clusters after the \( k^{th} \) position of B are renumbered as \( a+1, a+2, ..., \) and joined to A from the \( (k+1)^{th} \) position of A to form a child. The clusters after the \( k^{th} \) position of A are renumbered as \( b+1, b+2, ..., \) and joined to B from the \( (k+1)^{th} \) position of B to form the second child.
Case 2: The cut point cuts A at a border and B inside a cluster. Let \( p_1 \) and \( p_2 \) be the two parts that cluster \( b \) is broken into. \( p_1 \) is merged with the cluster \((a+1)\) of A and \( p_2 \) is merged with cluster \( a \) of A. The rest of the renumbering takes place as in Case 1.

Case 3: The cut point cuts A inside a cluster and B at a border. Let \( p_1 \) and \( p_2 \) be the two parts that cluster \( a \) is broken into. \( p_1 \) is merged with the cluster \((b+1)\) of B and \( p_2 \) is merged with cluster \( b \) of B. The rest of the renumbering takes place as in Case 1.

Case 4: The cut point cuts both A and B inside a cluster. Let \( p_1 \) and \( p_2 \) be the two parts cluster \( a \) is broken into. \( p_3 \) and \( p_4 \) be the two parts cluster \( b \) is broken into. \( p_1 \) is merged with the \( p_4 \) part of B and \( p_2 \) is merged with the \( p_3 \) part of B. The rest of the renumbering takes place as in Case 1.

If at least one of the two offspring is better than the parents in terms of fitness, the crossover is said to be successful. The two offspring take the place of the parents. Otherwise, the procedure is repeated \( r \) times where the value of \( r \) can be fixed during implementation. If not successful, the parents themselves are put back into the next generation. After crossover, the \( n/2 \) pairs of individuals give rise to \( n \) new individuals.

3.4.1.5 Genetic Operator: Mutation

Mutation is needed to counteract the loss of some potentially useful genetic material during selection and crossover. In an artificial chromosome, this is effected by an occasional random alteration of the value of a string position. In a binary implementation, a bit value is toggled. In an integer or floating point implementation, a value is changed within an allowed range. This definition cannot be directly applied to the present scenario.

A mutation operator that works at the boundaries of clusters has been worked out. For this, the pair-wise fitness between two consecutive data points is found for the whole data set. This has been calculated using the FFF and will be explained in the next section. Here, it is enough to understand that the pair-wise fitness value is a measure of the fitness of two consecutive data points if they formed a cluster of size two. The fitness value being in percentage, can vary from 0 to 100. The probability of mutation is a parameter
that can be changed. If the mutation probability is fixed as 0.05, then the number of clusters on which mutation has to be done will be \(0.05 * m_i\) (= nmutation) for an individual with \(m_i\) clusters. Following are the steps involved in mutation. Algorithm 3.5 gives the pseudocode for implementation of mutation.

1. In each individual, select randomly, points in the range \((1, m_i)\) where \(m_i\) is the number of clusters in the \(i^{th}\) individual. The number of clusters to be selected is equal to \(nmutation\) as worked out in the previous paragraph. Mutation is applied on the left as well as on the right border of each selected cluster which amounts to \(2*nmutation\) borders.

2. For a border point
   
   Two cut-off values (in %) are to be fixed, one below which the pair-wise fitness will be classified as insignificant (\(lval\)) and the other above which the pair-wise fitness will be significant (\(hval\)). The actual values will be implementation dependent.

   i) Mutation Rule 1: if its pair-wise fitness with the data point in the neighbouring cluster is greater than \(hval\) and its pair-wise fitness with the neighbouring data point within the same cluster is less than \(lval\), then reallocate the data item to the neighbouring cluster.

   ii) Mutation Rule 2: if its pair-wise fitness with the data point in the neighbouring cluster is greater than \(hval\) and also its pair-wise fitness with the neighbouring data point within the same cluster is greater than \(hval\), then the two clusters can be merged.

   iii) Mutation Rule 3: if its pair-wise fitness with the data point in the neighbouring cluster is less than \(lval\) and its pair-wise fitness with the neighbouring data point within the same cluster is also less than \(lval\), then the border point can be made a single-point cluster to which zero fitness value is assigned.

   iv) Mutation Rule 4: if none of the above conditions apply, then the border is left undisturbed.
An example of the process of mutation is given in Fig. 3.5. Here, the data item $g_4$ at the right border of the selected cluster can de-link and join the next cluster (mutation rule 1), the two clusters can be merged (mutation rule 2), it can become a single-unit cluster (mutation rule 3) or the two clusters may be left unchanged (mutation rule 4).

Fig. 3.5: Visualization of Mutation
Algorithm (Mutation)

begin
    initialize **// input data and variables**
    n = populationsize
    N = size of the data set
    mprob = 0.05 **// probability of mutation**
    m[i], i = 1, n **// number of clusters in the i\textsuperscript{th} individual**
    population[i], i = 1, N **// each element is an array of integers**
    pairfitness[i], i = 1, N-1

    nmutation = m[i] * mprob
    i = 0
    do
        i = i + 1 **// for all n members**
        j = 0
        do
            j = j + 1 **// for all clusters selected for mutation**
            k = select_random_number(1, m[i])
            apply mutation(i, k) **// apply mutation rules to left and right borders of k\textsuperscript{th} cluster**
        until
        j = nmutation
    until
    i = n

    return new population
end

Algorithm 3.5: Mutation

The cut off percentage value of pair-wise fitness for consideration as high/low can be changed. In short, mutation attempts to reallocate clusters at the borders, based primarily on the criterion of pair-wise fitness. This should improve the overall fitness of the individual.

This completes the discussion of all elements of the GA except the fitness evaluating function. Each of the traditional steps of a GA have been moulded to solve the clustering problem. Representation of a solution as a set of clusters, creation of the initial population of individuals where each individual is a different clustering possibility, selection of individuals with a set of better clusters and genetic operators that result in new clustering formats have been dealt with, in this section. But if all these have to work, the fitness of the clusters has to be evaluated in every iteration of the GA. This is where the fuzzy part comes in and the next section is devoted to how fuzzy logic has been used for fitness evaluation.
3.4.2 The Fuzzy Part

While the population of the genetic algorithm undergoes evolution at every generation, the relatively ‘good’ solutions reproduce while the relatively ‘bad’ solutions die. To distinguish between solutions, an objective (evaluation) function is used. In the simple cases, there is only one criterion for optimization for example, maximization of profit or minimization of cost. But in many real-world decision making problems, there is a need for simultaneous optimization of multiple objectives.

3.4.2.1 Why fuzzy ?

A single objective optimization model cannot serve the purpose of a fitness measuring index because we are looking at multiple criteria that could be responsible for stringing together data items into clusters. This is true, not only for the clustering problem but for any problem solving using GA that involves multiple criteria. In multi-criteria optimization, the notion of optimality is not clearly defined. A solution may be best w.r.t. one criterion but not so w.r.t. the other criteria. Pareto optimality (Chapter 2) offers a set of non-dominated solutions called the P-optimal set where the integrity of each of the criteria is respected.

We can think of a P-optimal set at the individual (a set of clusters) level where each element of the P-optimal set is an optimum clustering solution w.r.t. a particular criterion. But, the problem here, calls for a partitioning of the sequential data set into a set of clusters where each cluster is separately optimal w.r.t. the multiple criteria. Therefore, the criteria are not acting at the individual (set of clusters) level but at the cluster level. So, though the GA-based clustering algorithm requires a multi-objective optimization, the multi-objective GA (MOGA) using the pareto optimality approach cannot be applied. What is required is a method for aggregation of the contribution of each criterion at the cluster level.

In the classical method of objective weighting, each criterion is given a weight and the weighted sum of the fitness of individual criteria is taken as the overall objective function. The weights are a measure of the significance of a criterion in comparison with the other criteria. For example if one weight is two times the other, it implies that
the former criterion is doubly significant compared to the latter. However, a set of fixed
weights leads to a constant interrelationship. If values for one or more of these criteria are
not available, then the total fitness score will drop drastically. Actually, some of the
other criteria may be enough to give a high combined score. Under different
combinations of inputs, the weight values can vary. It is here that fuzzy logic comes to
the rescue for it specializes in comparing apples and oranges.

In a fuzzy-logic based system, a weighting function is replaced by a set of rules. A
criterion can take a whole range of values in an interval. The criterion, if modeled as a
fuzzy variable can belong to different classes in the interval with different probabilities.
Different combinations of different criteria (fuzzy inputs) join together to produce unique
effects. A fuzzy logic inferencing system can capture the complex interactions among
the criteria which is not possible by assignment of fixed weights to the inputs.

3.4.2.2 The Fuzzy Fitness Finder

The GA calls upon the Fuzzy Fitness Finder (FFF) to evaluate the fitness of the solutions
it creates in each population. A solution is a mapping of the whole data set into clusters.
A cluster may have one or more data objects. The fitness of an individual is calculated as
the summation of the fitness values of all its constituent clusters whose size is greater
than one. A single unit cluster is assumed to have fitness equal to zero.

\[
\text{Fitness} = \sum_{i=1}^{m} \text{fitness}_i
\]

where \( m \) = number of non-single clusters in an individual and \( \text{fitness}_i \) is the fitness of
the \( i^{th} \) cluster of the individual. To find the values for \( \text{fitness}_i \), a fuzzy inferencing
mechanism has been developed.

The FFF is used in two ways -

i) to find the pair-wise fitness between adjacent data items

ii) to find the fitness of a set of data items (a cluster)

The same FFF can be applied to both cases. In the first case, the cluster size is equal to
two and as it is between adjoining data items of the data set, these values can be
calculated and stored at the outset. It may be recollected that these values are used by the mutation operator to adjust the cluster borders. The FFF calculates the fitness values of all non single clusters of individuals in the initial population as well as in subsequent generations.

Number of calls to FFF for calculating the fitness of the population = \[ \sum_{i=1}^{n} m_i \]

where \( m_i \) is the number of non single clusters in the \( i^{th} \) individual and \( n \) is the size of the population.

The fitness calculations are done for clusters of the initial population. In each new generation, variation is brought about by crossover and mutation. For crossover, only if the cut-point falls inside a cluster, the fitness values have to be recalculated for the disturbed clusters. Otherwise, it involves only a renumbering of clusters from the cut-point till the end of the data set. Fitness values have to be re-calculated for clusters affected by mutation. Whenever cluster constitution changes, the FFF is called to recalculate the new cluster’s fitness value. The FFF is a standard fuzzy logic based inference system used in engineering and control systems.

The elements of the FFF are

- Identification of input and output criteria
- Calculation of crisp values of the input criteria
- Fuzzification of input values
- Fuzzy Inference Engine
- Defuzzification of output values

Each of these elements will be discussed in detail. A schematic model of the FFF is shown in Fig. 3.6.
3.4.2.3 Input and Output Criteria

The various criteria that influence the formation of clusters are taken as the input variables. The number of input criteria will be greater than or equal to one. The predominant criterion is used to form the initial population but can also be included as an input criterion. The values that the input variables take are the values pertaining to the input criteria which in turn, may be in numeric form, as textual information or may be already in a fuzzy form. The output variable in the algorithm is only one, namely, the fitness value of the cluster in percentage. So, the system is a multiple input single output system.
3.4.2.4 Crisp values for the Input Criteria

The input criterion may be the presence or absence of a common feature between any two data points in the cluster, or the distance between consecutive data items in the cluster. Some amount of preprocessing has to be done in order to transform this information about a criterion operating within the cluster into a single crisp value for further operations in the FFF.

For every cluster, the values that input variables take have to be derived. The information available on the input criterion can be such that:

i) it relates two adjacent data items in the cluster or
ii) any two data items in the cluster.

A measure of the relation between any two data points can be numerical, boolean or a multivalued logical value etc. These values have to be transformed into a crisp value.

For numeric values like distance between adjacent data items, the average of all values can be taken as the crisp value. For Boolean values, if two data items have a feature in common, then score for that pair is taken as 1 else it is assumed to be 0. The sum of these pair-wise scores is divided by the total number of combinations to give a crisp value in the range (0,1). For a cluster of size m, the total number of combinations will be \((m-1)\) if only adjacent comparisons are made or \(^nC_2\) if all pairs are averaged.

3.4.2.5 Fuzzification of Inputs

The first step for absorption of the input data by a fuzzy system is to fuzzify it. A fuzzy variable may be considered as a set of fuzzy sets. Fuzzifying an input data value involves the evaluation of all the membership functions of the variable. The result of evaluation is a vector of membership values. This process is repeated for all the input variables.

A fuzzy set is fully defined by its membership function. For most control applications the sets that have to be defined are easily identifiable. For other applications they will have to be determined by knowledge acquisition from an expert. Once the names of the fuzzy sets have been established the associated membership functions have to be designed.
Linear triangular fuzzy sets are the easiest to implement. But for many applications, triangular membership functions cannot represent accurately the linguistic terms being modeled. They have to be elicited directly from the expert by a statistical approach or by automatic generation with available training data and no contribution from the expert.

For this implementation of the algorithm, we create the fuzzy sets based on heuristics by dividing the domain of discourse into triangular fuzzy sets with 10% to 20% overlap typically. A triangle membership function can be defined by a triad \((a,b,c)\). \(a\) and \(c\) are the endpoints and the center point \(b\) can be assumed to be the center of gravity of the group. The peak of the triangle has membership equal to one. Let \(I\) be an input variable that takes values in the range \([0,1]\). Table 3.1 shows one of the many possible ways to split the range of discourse \([0,1]\) into fuzzy sets along with their linguistic interpretations and the values of the triads \((a,b,c)\) that define the fuzzy sets. Fig. 3.7 shows the corresponding membership functions.

![Fig. 3.7: Fuzzy Memberships for an input variable with range [0,1]](image)

<table>
<thead>
<tr>
<th>Linguistic Definition</th>
<th>Lower Limit (a)</th>
<th>Peak Value (b)</th>
<th>Upper Limit (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>V1</strong>: Very Low Similarity</td>
<td>0.0</td>
<td>0.0</td>
<td>0.25</td>
</tr>
<tr>
<td><strong>V2</strong>: Low Similarity</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td><strong>V3</strong>: Medium Similarity</td>
<td>0.35</td>
<td>0.5</td>
<td>0.75</td>
</tr>
<tr>
<td><strong>V4</strong>: High Similarity</td>
<td>0.7</td>
<td>0.8</td>
<td>0.9</td>
</tr>
<tr>
<td><strong>V5</strong>: Very High Similarity</td>
<td>0.85</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 3.1: Fuzzy sets for an input variable with range \([0,1]\)
If there are \( k \) criteria, an average of \( m \) number of clusters in an individual and \( n \) individuals in the population, Algorithm Fuzzify has to executed \( k \times m \times n \) times for the initial population. The pseudocode of the algorithm Fuzzify is given in Algorithm 3.6

**Algorithm (Fuzzify)**

```
begin
  initialize // input data and variables
  \( k \) = number of input variables
  \( m[i], i = 1, k \) // number of fuzzy sets of the \( k \)th variable
  \( \text{inpval}[i], i = 1, k \) // \( k \) input values
  \( \text{lowval}[i][j], i = 1,...,k, j = 1,...,m[i] \) // lower limits of fuzzy sets
  \( \text{midval}[i][j], i = 1,...,k, j = 1,...,m[i] \) // peak values of fuzzy sets
  \( \text{highval}[i][j], i = 1,...,k, j = 1,...,m[i] \) // upper limits of fuzzy sets
  \( \text{memb}[i][j], i = 1,...,k, j = 1,...,m[i] \) // \( \text{memb}[i][j] \) membership of \( i \)th variable in its \( j \)th fuzzy set
  \( i = 0 \)
  do \( i = i + 1 \) // for all \( k \) variables
    \( j = 0 \)
    do \( j = j + 1 \) // for each fuzzy set
      if ( \( \text{lowval}[i][j] < \text{inpval}[i] < \text{highval}[i][j] \) ) then
        if ( \( \text{inpval}[i] \leq \text{midval}[i][j] \) ) then
          \( \text{memb}[i][j] = (\text{inpval}[i] - \text{lowval}[i][j]) / (\text{midval}[i] - \text{lowval}[i][j]) \)
        else
          \( \text{memb}[i][j] = (\text{highval}[i] - \text{inpval}[i][j]) / (\text{highval}[i] - \text{midval}[i][j]) \)
      until \( j = m[i] \)
    until \( i = k \)
return \( \text{memb}[][] \) // membership functions
end
```

Algorithm 3.6: Fuzzify

The triads of the fuzzy sets of each variable are stored in arrays lowval, midval and highval. If the input data value (inpval) falls within a fuzzy set, then it will have positive membership value which is calculated from the triad values of that set otherwise the membership value remains zero. Membership values lie in the range \((0,1)\). The \( i \)th row of the array \( \text{memb}[i][] \) contains all the membership values corresponding to the \( i \)th input variable.
Fig. 3.8: Fuzzification of an input

Fig. 3.8 illustrates how fuzzification works. Let the input criterion take a value $x$. It belongs to both fuzzy sets $A_i$ and $A_{i+1}$ with membership values equal to 0.75 and 0.25 respectively. The membership in all other sets will be zero.

### 3.4.2.6 Fuzzy Inference Engine

The fuzzified values for the different input criteria are now available. Each criterion belongs to one or two fuzzy sets with non-zero probability depending upon whether it belongs to non-overlapping or overlapping part of fuzzy sets. The primary aim of the fuzzy inference engine is to combine the effects of various criteria available as fuzzy inputs to calculate a fitness value for the cluster using the fuzzy rules.

The Fuzzy Inference Engine can be considered as a black box into which are fed the outputs of the fuzzifier. The main steam is derived from the fuzzy rule base that is stored in the data bank of the inference engine. In industrial applications, the inputs and outputs are defined in terms of fuzzy variables and a set of IF-THEN rules are constructed to decide what actions to take under different input conditions. The inputs are generally physical quantities like temperature, volume etc. and the output is an action to be taken like the setting of a heat knob or how much a valve should be opened.

In the case of the clustering algorithm of our interest, the inputs and outputs are not physical quantities. The inputs modeled as fuzzy variables are the $k$ criteria influencing...
the cluster formation. The output is a value that can be taken as a measure of the fitness of the cluster. A hypothetical example has been taken to show the format of a rule.

Number of Criteria = 2

Criterion 1 can take values in fuzzy sets Less, Medium and High.

Criterion 2 can take values in fuzzy sets Small, Big and Large

Let the output (fitness) take values in fuzzy sets Poor, Satisfactory, Good and Excellent.

Rules in the rule set are a combination of two antecedents and a consequent. Some examples of rules have been given.

**IF** criterion1 is High **AND** criterion2 is Large **THEN** fitness is Excellent

**IF** criterion1 is Medium **AND** criterion2 is Large **THEN** fitness is Good

**IF** criterion1 is Less **AND** criterion2 is Small **THEN** fitness is Poor

The basic logic operators used to compose rules are the AND, OR and the NOT operators. Here, only the AND operator has been used as the stress is on combining the criteria. But the other possibilities remain open to the designer.

The fuzzy rule base plays the most important role in the fuzzy inferencing mechanism. In many real-life applications, rules are either determined by experts or are extracted from a given input-output data set. When applied to clustering, there could be observed data available where clusters are known and in these clusters, the contribution of the input criteria can be easily calculated. Rules can be formulated from this input-output data. However, this is possible only in a small subset of problems. So, human judgment on the relative importance of each criterion has to be used to construct the rules. Also the output variable is a fitness value which is not a physical quantity and the values can only be decided rationally. Hence, like the choice of the input fuzzy sets, the output fuzzy sets and the fuzzy rulebase are also constructed intuitively.

The maximum number of possible rules is simply the product of the number of fuzzy sets for each variable, but definition of all of these rules may not be necessary since some input conditions may never occur in reality. The rule set should map out the universe of possible input combinations while keeping the system sufficiently under control. In a real
life situation, some of the combinations of inputs may never materialize. If there is
definite information about certain impossible situations, then rules need not be framed for
these cases. As we do not have any real training data, it becomes necessary to cover the
entire product space of input combinations.

The inputs (making the antecedents) and output (consequent) to the fuzzy rule base are
fuzzy variables. Every crisp input for each criterion will have non-zero memberships in
one or more than one fuzzy set due to the overlap of the triangles. This leads to many
antecedents of rules getting activated which cause different output fuzzy sets to fire.
Algorithm 3.7 gives the pseudocode for the fuzzy inference engine.
Algorithm (Fuzzy Inference Engine)

begin
initialize // input data and variables
k = 3 // number of input variables
m[i], i = 1, k // number of fuzzy sets of the kth variable
memb[i][j], i = 1,...,k, j = 1,...,m[i] // memb[i][j] is the membership of i
isetname[i][j], i = 1,...,k, j = 1,...,m[i] // input set names
// linguistic code e.g. HIGH, LOW
antecedents[i], consequent[i], i = 1,...,r // r rules
// antecedents concatenated e.g. HIGHLOW
// output e.g. LOW
osetname[i], i = 1,...,q // output set name
osetvalue[ii] = 0, ii = 1,...,q // output strength
i = 0
do i = i + 1 // for first variable
j = 0
do j = j + 1 // for second variable
k = 0
do k = k + 1 // for third variable
// look at all combinations
if (memb[1][i] > 0 and memb[2][j] > 0 and memb[3][k] > 0) then begin
addsets = isetname[1][i] + isetname[2][j] + isetname[3][k]
p = 0
do p = p + 1
if (addsets = antecedents[p]) then begin
poutput = min(memb[1][i], memb[2][j], memb[3][k])
if (consequent[p] = osetname[ii]) then //ii = 1,...,q
osetvalue[ii] = osetvalue[ii] + poutput*poutput
end
until p = r
end
until k = m[3]
until j = m[2]
until i = m[1]
osetvalue[ii] = sqrt(osetvalue[ii]) //ii = 1,...,q
end
The logical products for each rule must be combined before being passed on to the defuzzification process for crisp output generation. Several inference methods exist. The root-sum-square (RSS) method has been used to combine the effects of all applicable rules.

For explanation, the case of three input variables has been taken. The variables have $m[1]$, $m[2]$ and $m[3]$ fuzzy sets respectively. The total number of rule combinations possible are $m[1]*m[2]*m[3]$. The rule base is stored in two arrays each of size $r$. There are $r$ antecedents and $r$ consequents. The antecedent[i] gives rise to consequent[i] where $i$ lies in (1,$r$). The antecedent array is an array of strings. Each element is a concatenation of the input set names and the corresponding consequent is also a string which is the output set name. The firing strength of a rule is equal to the minimum of the membership values of the 3 variables in their fuzzy sets $(i, j, k)$ that the loop is executing at that instant. These were generated by the ‘fuzzifier’ routine. Whenever a rule fires, the corresponding output set value is updated by adding the square of the firing strength (sum of squares). For example, if the consequent of the fired rule is the 2nd fuzzy set, then osetvalue[2] is updated. At the end, the square root of the elements of osetvalue is taken (root sum of squares). The output of the fuzzy inference engine is the array of output set values stored in the array osetvalue $[ii]$ , $ii = 1, \ldots, q$ where $q$ is the number of output fuzzy sets.

The antecedent part of the rule takes all the criteria together. This system is easy to implement if the number of criteria are not more than three and the number of input fuzzy sets for each criterion are not many. However, combining more criteria implies an exponential increase in the number of rules making the formulation of the rule base difficult. In the application of the algorithm which follows in the next chapter, the four criteria case was handled differently. An alternate method of splitting the rule base was used. The inference engine works in a cascading fashion (or pipeline). Two criteria are combined to give an output fitness value using a rule base with only two antecedents. The output is combined with the next criterion and put through the inference engine again. Thus, the inference engine is called three times to get the cluster fitness based on four criteria. As only two inputs are handled at a time, the rule bases are simpler to create. For example, if there are four input variables, each having nine fuzzy sets, then the total
number of rules when all the variables are taken together will be \(9^4 = 6561\). But taken in the cascading mode, there will be three rule bases each of size \(9^2 = 81\) rules. The three rule bases would have a total of only 243 rules. However, equivalence between the two methods has to be ensured by the designer, i.e. if four inputs \(I_1I_2I_3I_4\) give output \(O\), then \(I_1I_2\) gives output \(O_1\) followed by \(O_1I_3\) giving output \(O_2\) which together with the last input \(O_2I_4\) should give the same output \(O\). This method is easy to scale up to add more criteria and is therefore, free of the combinatorial explosion problem.

### 3.4.2.7 Defuzzification of Output

Defuzzification means getting back a crisp value for the output. The standard methods of defuzzification were introduced in Chapter 2.

The defuzzification of the data into a crisp output is accomplished by combining the results of the inference process and then computing the "fuzzy centroid" of the area. The output fitness value (O) for a cluster is given by

\[
O = \frac{\sum_{i=1}^{q} w_i b_i}{\sum_{i=1}^{q} w_i}
\]

where \(w_i\) is the weighted strength and \(b_i\) is the centre point of the \(i^{th}\) output membership function and \(q\) is the total number of output membership functions. The values for \(w_i\) are stored in the array osetvalue[] which is the output of the fuzzy inference engine. The weighted strengths of each output membership function are multiplied by their respective output membership function centre points and summed. Finally, this area is divided by the sum of the weighted membership function strengths and the result is taken as the crisp output. The output is a percentage value. Algorithm 3.8 gives the algorithm for defuzzification.
Algorithm (Defuzzify)

begin
    initialize // input data and variables
    // q = number of output membership functions
    osetvalue[i], i = 1,..., q // output membership function strengths
    b[i], i = 1,...,q // centre values of the output membership functions
    SumStrengths = 0 // sum of membership function strengths
    Output = 0 // Crisp output fitness value of the cluster
    i = 0
    do i = i + 1 // for all q values of output membership functions
        Output = Output + osetvalue[i] * b[i]
        SumStrengths = SumStrengths + osetvalue[i];
    until i = q
    Output = Output / SumStrengths
end

Algorithm 3.8: Defuzzification

3.4.3 The Evolutionary Process

The search for better clusters continues by mixing clusters from individuals and shifting the borders of clusters until a terminating condition is satisfied. The terminating condition can be an upper limit for the generation counter or until there is no significant change in average fitness of the population.

The choice of the initial population plays a major role in the course that the evolutionary process takes. If the ancestors consist of very bad clusters, the crossover operator would not be able to improve the lot of the offspring as it will be only cutting and joining the not-so-good clusters. Then, the onus will be on the mutation operator to change the cluster constitution. The ancestors are chosen by clustering with different cut-off values. Generally, the designer will have an idea of what range these values would lie in. This is highly problem-specific. An intelligent choice of the cut-off values can go a long way in making the evolutionary process start off in the right direction.
Genetic algorithms are sensitive to control parameters like population size, probability of crossover and mutation, the number of times crossover has to be repeated before it is successful and value of the generation counter or the number of iterations.

In the fuzzy module, the rule antecedents, consequents and the variables of the input and output membership functions can be changed. Adding additional degrees (or fuzzy sets) to the input and/or output functions like “really high” or “really low” can also change the performance of the fuzzy system and hence the results of the GA.

Soft computing techniques are sensitive to control parameter selection. Selection of control parameters is problem dependent. For each problem, one has to tune the parameter values to suit the application. Tuning the system amounts to "tweaking" the defining parameters to achieve acceptable system response.

### 3.5 Performance of the Algorithm

Any clustering algorithm results in clusters. To determine how good a clustering scheme is, there are several standard indices defined. The Davies-Bouldin (DB) index has been modified to suit the proposed algorithm. Time and space complexity are two important performance indicators. An estimate for these parameters have been calculated.

#### 3.5.1 Cluster Quality Index

**The Davies-Bouldin Index**

The Davies-Bouldin (DB) index as described in Chapter 2, is a function of the ratio of the sum of “within-cluster” scatter to “between-cluster” separation. The Euclidean distances between cluster centres is a measure of the separation while deviation from the cluster centre within clusters is a measure of the scatter. The ratio of the scatter to separation is averaged over all the cluster combinations. Minimization of the DB value leads to better clustering.
The Modified Davies-Bouldin Index

Since the proposed clustering scheme is based on fuzzy fitness scores and not on Euclidean distances, the DB analysis was suitably modified to determine the equivalent of “within-cluster” scatter and “between-cluster” separation. The clusters being contiguous, between-cluster separation needs to be taken only with the two neighbour clusters, one on the right side and the other on the left side. In case of the first cluster and the last cluster, only one side comparison can be made, namely, the separation from the right neighbour and that from the left neighbour respectively.

Cluster Quality Index, \( CQI = \frac{1}{n_c} \sum_{i=1}^{n_c} R_i \)

where \( n_c \) = number of clusters

\[
R_i = \max_{j=i-1}^{j=i+4} \frac{S_i}{d_{ij}}
= \max_{j=i-1}^{j=i+4} \frac{100-f_i}{100-p_{ij}}
\]

where \( S_i \) is the scatter within the \( i^{\text{th}} \) cluster,
\( d_{ij} \) is the separation between the \( i^{\text{th}} \) cluster and the \( j^{\text{th}} \) cluster.

The maximum is taken over the left and right ratio values. The fuzzy fitness score \( (f_i) \) for the cluster is a measure of the similarity of the genes constituting the cluster with respect to the multiple features compared. Thus a high fitness score is equivalent to a low scatter within the cluster and vice versa. Therefore \( (100-f_i) \) is considered equivalent to the scatter \( S \) in our calculation. Similarly the separation between two clusters \( d_{ij} \), is taken as \( (100-p_{ij}) \) where \( p_{ij} \) is the pair-wise fitness with the bordering clusters. Therefore, more the pair-wise fitness at the border, lesser will be the separation between the two clusters.
3.5.2 Time and Space Complexity

3.5.2.1 Time Complexity

An expression for time complexity has been derived as follows.

Time Complexity = Complexity (initial population)  
+ Complexity (an iteration) * number of iterations

Let \( n_{\text{iter}} \) = number of iterations of the algorithm  
\( n \) = population size or number of individuals  
\( N \) = total number of data items in an individual  
\( m \) = average cluster size excluding single unit clusters  
\( n_f \) = number of features  
\( k \) = average number of non-single clusters in an individual  
\( K \) = average number of clusters in an individual including single clusters  
\( m_p \) = average number of clusters affected due to mutation in an individual.

Complexity (initial population) = Complexity (creating clusters)  
+ Complexity (computing fitness values of the clusters)  
\[ = n.N + \binom{m}{2}n_f + n_f - 1)k.n \]
\[ = O(n.N + \binom{m}{2}n_f.k.n) \]

(1)

If data items in a cluster have to be compared pair-wise then \( \binom{m}{2} \) comparisons have to be made to arrive at a crisp value for each feature. For \( n_f \) number of features, the fuzzy fitness finder is executed \((n_f - 1)\) times for each cluster. \((n_f - 1)\) can be ignored as \( \binom{m}{2}n_f > (n_f - 1) \) since \( \binom{m}{2} \geq 1 \) (as \( m \geq 2 \)).

Complexity (an iteration) = Complexity (selection + crossover + mutation)  
\[ = n + \binom{m}{2}n_f.n + \binom{m}{2}n_f.m_p.n \]
\[ = O(\binom{m}{2}n_f.m_p.n) \]

(2)
Crossover can be performed a maximum of $r$ times. If not successful, then the parents are put back into the new population. Time gets multiplied by a factor of $r$ at most but being a constant multiplier, it can be ignored for the calculation of time complexity.

From (1) and (2), an estimate of the time complexity is hence given by $O(n.N^mC_2 \cdot n_f \cdot n(k + n_{iter} \cdot m_p))$. The term $^mC_2$ will be replaced by $m$, if only adjacent members of a cluster are compared.

For the first generation, all clusters in each individual are evaluated for their fitness. Hence the main computational burden is incurred at the start. In following iterations, crossover involves only a recombination of clusters. Only the two clusters that are affected by crossover have to be recalculated. Mutation too involves only the re-calculation of altered clusters.

### 3.5.2.2 Space Complexity

Space Complexity can be calculated by limiting to quantities relating to the number and magnitude of the inputs and outputs of the algorithm. There is space required to store information on the data items and information about the clusters.

Space Complexity = Space for data items + Space for cluster information.

Every data item has criteria information. For every individual, each data item will have a cluster number.

Space for data items = $n_f.N + n.N$  \hspace{1cm} (3)

Pertaining to each cluster, there are $n_f$ scores for criteria values and one value for cluster fitness. For storing the values of the whole population, the space needed has to be summed over all clusters of an individual and then over all the individuals.

At every instant, the values of the new population are being calculated from the old population. Hence the same space is needed for the new population also.

Space for cluster information = $2K.n.(n_f + 1)$  \hspace{1cm} (4)

Adding (3) and (4), an estimate of space complexity is given by

$$S_p = N(n_f + n) + 2.K.n(n_f + 1)$$
3.6 Comparison with other clustering algorithms

The various clustering algorithms and their variations were reviewed in Chapter 2. A comparison of these techniques was also done in Chapter 2. Here, we look mainly at the time complexities of the algorithms.

The time complexity of the popular k-means algorithm is \(O(krN)\) where \(k\) is the number of desired clusters, \(r\) is the number of iterations and can be large, \(N\) is the size of the data set that can be really large. The shortest spanning path algorithm that takes a graph theoretic approach has complexity \(O(N^2)\). The worst case time complexity of hierarchical complete-link clustering algorithm is at most \(O(N^2 \log N)\). The time complexity of hierarchical single-link clustering is \(O(N^2)\).

From the time complexity point of view, the k-means algorithm seems to be the best. In the case of the fuzzy guided genetic algorithm, the complexity depends on many parameters like the size of the data set (\(N\)), population size (\(n\)), the size of the clusters (\(m\)), the number of clusters (\(k\)), number of iterations and probability of mutation. The cluster fitness calculation for every cluster involves the maximum number of computations. At the start, the fitness of \((n*k)\) clusters has to be calculated, where \(n\) is the population size and \(k\) is the average number of non-single clusters in an individual. Generally, \(n\) does not take a very large value. \(k\) is a function of the data set (\(N\)). After the initial fitness calculations, the time complexity depends on the number of iterations and the number of clusters affected by crossover and mutation. One cannot make a direct comparison between clustering algorithms as each has its own strengths and weaknesses. Though, it doesn’t improve the time complexity, the main power of the algorithm lies in its decision making capability based on multiple criteria.

GA perform a globalized search for solutions whereas most other clustering procedures perform a localized search. In a localized search, the solution obtained at the ‘next iteration’ of the procedure is in the vicinity of the current solution. In this sense, the k-means algorithm, fuzzy clustering algorithms, ANNs used for clustering, various annealing schemes, and tabu search are all localized search techniques. In the case of GA,
the crossover and mutation operators can produce new solutions that are completely different from the current ones. In the FGA, the genetic operators are designed to suit the problem of sequential data clustering. So, while crossover can produce clustering solutions that are very different from the parents, the action of mutation is localized by definition.

3.7 Conclusions

GA are blind search algorithms that exploit the coding and objective function value to find a solution. The crossover and mutation operator used in this algorithm combine a knowledge directed approach with the conventional evolutionary process. The knowledge comes in the form of the fuzzy decision making system. The combination of the two paradigms of soft computing, namely, GA and fuzzy reasoning gives a powerful tool for multi-criteria optimization which is otherwise difficult in a conventional GA.

As can be seen from the applications studied in Chapter 5, intuitively constructed fuzzy sets and rules using domain knowledge do make good predictions. But the present model can be strengthened if the data available on the multiple features can be used to generate the membership functions. Automatic generation of membership functions will be taken up in Chapter 6.