This study evaluates the efficacy of the continuous GA algorithm in solving two complex problems. The queuing network design problem has random processes and the decisions are subject to a cost constraint. Therefore, a simulation program is run to determine the fitness of each solution in population. The second problem is a large-scale gene-clustering problem in which thousands of genes have to be grouped into a fixed, small number of clusters on the basis of similarity. Clustering of gene expressions reduces the unmanageable volume of data into small number of sets that are of great interest to the biologists. For these problems, the GA algorithm produces good results compared to the alternative methods. The GA can also handle the same class of problems without the restrictive assumptions.
Genetic Algorithm for Continuous Variable Optimization with Applications to Queuing Network and Gene-Clustering Problems

A Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of System Analysis
Department of
Computer Science & System Analysis
by
Mouin Hourani
Miami University
Oxford, Ohio
2004

Adviser __________________
Dr. Mufit Ozden

Reader __________________
Dr. Frank W. Moore

Reader __________________
Dr. Pedrito U. Maynard-Zhang
## Table of Contents

Table of Contents ................................................................................................................ ii
List of Tables ........................................................................................................................ v
List of Figures ......................................................................................................................... vi
Chapter 1 ............................................................................................................................... 1
   Introduction ........................................................................................................................ 1
Chapter 2 ............................................................................................................................... 4
   Evolutionary Computation and Genetic Algorithm ....................................................... 4
      2.1 A Brief History of Evolutionary Computation .................................................... 5
      2.2 Biological Terminology ...................................................................................... 6
      2.3 Search Spaces ...................................................................................................... 7
      2.4 Genetic Algorithm ............................................................................................. 7
         2.4.1 Encoding ....................................................................................................... 9
            2.4.1.1 Binary Encoding .................................................................................. 9
            2.4.1.2 Permutation Encoding ...................................................................... 9
            2.4.1.3 Value Encoding .................................................................................. 10
            2.4.1.4 Tree Encoding .................................................................................... 10
         2.4.2 Evaluation .................................................................................................... 11
         2.4.3 Selection ....................................................................................................... 11
            2.4.3.1 Roulette Wheel Selection ................................................................... 11
            2.4.3.2 Rank-Based Selection ........................................................................ 12
            2.4.3.3 Steady State Selection ....................................................................... 12
         2.4.4 Crossover ..................................................................................................... 13
            2.4.4.1 Single Point Crossover ....................................................................... 13
            2.4.4.2 Two Points Crossover ....................................................................... 13
            2.4.4.3 Uniform Crossover ........................................................................... 13
            2.4.4.4 Arithmetic Crossover ....................................................................... 14
         2.4.5 Mutation ....................................................................................................... 14
            2.4.5.1 Value Mutation .................................................................................. 14
            2.4.5.2 Micro Mutation ................................................................................ 14
            2.4.5.3 Add Value Mutation ........................................................................ 14
         2.4.6 Parameters of GA ......................................................................................... 15
            2.4.6.1 Crossover Probability ....................................................................... 15
            2.4.6.2 Mutation Probability ......................................................................... 15
            2.4.6.3 Population size ................................................................................. 15
            2.4.6.4 Elitism Number ............................................................................... 15
         2.4.7 Random Number Generator ....................................................................... 16
Chapter 3 ............................................................................................................................. 17
   Genetic Algorithm Applications to Two Problems ...................................................... 17
      3.1 Queuing Network Problem ................................................................................. 18
         3.1.1 Problem description .................................................................................... 19
         3.1.2 Cost Functions ............................................................................................ 21
         3.1.3 Maximization of the Throughput Based on BFS .................................... 22
List of Tables

2.1 Fitness Value and Selection Probability for Roulette Wheel Selection----- 11
2.2 Fitness Value and Selection Probability for Rank-Based Selection-------- 12
2.3 Fitness Value and Selection Probability for Steady State Selection with 50%
------------------------------------------------------------------------------- 12
3.1 The Cost Factors ci and Exponents αi for Queuing Network Components 23
3.2 The Optimal Result after Applying BFS---------------------------------------- 23
3.3 The Average Throughput for Queuing Network with the Optimal Parameter
Settings of the BFS ------------------------------------------------------------- 25
3.4 The Average Throughput Estimated for Queuing Network with the GA -- 28
3.5 Comparison of GA and BFS Results. ------------------------------------------ 29
3.6 Clustering Results after Applying GA. ---------------------------------------- 36
3.7 Clustering Results after Reordering Centroids in the GA --------------------- 38
3.8 Clustering Results after Applying GA on 20% a Sample Data. ----------------- 39
3.9 Clustering Results after Applying GA on 20% a Sample Data. ----------------- 40
3.10 Clustering Results after Applying GA on 20% a Sample Data. ----------------- 40
3.11 Clustering Results after Applying GA with Recalculating Centroid. ------- 43
3.12 K-means Clustering Results for Test Data ----------------------------------- 45
3.13 Applying GA with the Fourth Approach to Gene Expression Data ----------- 46
3.14 K-means Clustering Results for Genes Expression --------------------------- 47
3.15 Comparison of the Adaptive GA and Known Solution ------------------------ 48
List of Figures

2.1 Skeleton of Genetic Algorithm----------------------------------------------- 8
2.2 Binary Encoding Example----------------------------------------------------- 9
2.3 Permutation Encoding Example ----------------------------------------------- 10
2.4 Examples of Different Types of Value Encoding ----------------------------- 10
2.5 Examples of Different Types of Tree Encoding ----------------------------- 10
2.6 Single Point Crossover------------------------------------------------------ 13
2.7 Two Points Crossover-------------------------------------------------------- 13
2.8 Uniform Crossover----------------------------------------------------------- 13
2.9 Arithmetic Crossover-------------------------------------------------------- 14
3.1 Queuing Network-------------------------------------------------------------- 21
3.2 T-Distribution--------------------------------------------------------------- 27
3.3 K-means Algorithm Skeleton--------------------------------------------------- 32
3.4 An Example of K-means-------------------------------------------------------- 32
3.5 Data Points with Fifteen Clusters Projected on 2D-------------------------- 35
3.6 An Example for Crossover without/with Centroid Reordering ----------------- 37
3.7 The Relationship between Average Running Time and Sampling Percentage -------------------------------------------------- 41
3.8 The Relationship between Average Percentage of Correctly Assigned Points and Sampling Percentage----------------------------------- 42
3.9 Run Times vs. Fitness Measures of Four Approaches-------------------------- 44
Acknowledgments

I would like to express my sincere appreciation to my adviser Dr. Mufit Ozden for his guidance and encouragement in my graduate study. His bright advice shaped my thesis into a more precious work. His friendly attitude made the difficulties easier to overcome.

I would also like to express my thanks to my thesis committee, Dr. Frank Moore and Dr. Pedrito Maynard-Zhang, for their time and helpful advice.

I will not forget my good friends, brother and sister Ahmet S. Kalafat and Hanife Baskan, for their help and support.

Special thanks go to my friend Christopher Peterson for his help.

I deeply appreciate my mother, Siham Olabi, for the loving support she gave me through all of my endeavors.

Big thanks are also due to my siblings Tarif, Imad, Awni, and Mona Hourani, for their help and support.

I greatly appreciate Samaa Moussalli, my wonderful wife, for her support, inspiration, patience and love in my good times and in my bad times.

And finally, I hope for a great future for my beloved son, Ridwan Hourani, whom I had to stay far from throughout my graduate study.

Without above people, this thesis and degree would have remained only a dream.
Chapter 1

Introduction

Genetic Algorithms (GA) are a form of evolutionary computation, which is an important sub-discipline of machine learning. GAs are a search technique inspired from the biological process of evolution by means of natural selection. They can be used to construct a numerical optimization technique that performs effectively on problems characterized by huge search spaces.

In GAs, the variables that we are trying to optimize must be encoded in one of several different ways. Binary, integer, real, and complex structure encoding (e.g., a tree) are among the most common examples. The most famous encoding is Binary Encoding because the initial research on GA used this type of encoding. The binary encoding is relatively simple to implement. This study develops the continuous variable GA strategy to work with the two complex problems, namely Queuing Networks and Gene Clustering. The scope of this study covers only the continuous variable cases of these two particular problems.
The first application is a queuing network problem. A queuing network is a set of interconnected queuing systems. Queuing networks are commonly used for the performance and reliability evaluation of computer communication systems, manufacturing systems, airport check-in systems, fast food restaurants, and many other service-based applications.

In many queuing network design problems, there may be several decision variables, such as service rates, routing probabilities and buffer sizes. These variables have to be determined optimally in terms of a system performance (e.g., throughput). There may also be some constraints on these variables (e.g., a cost function).

The GA algorithm employs the fitness evaluation of each candidate solution in order to select a subset of these solutions for “breeding”. To provide a fitness value for a queuing network design solution, we need to run a simulation program for this network design due to the random interaction of system components. The simulation program uses random variables to assign a service time for the stations of the network and/or arrival rates for the entities. Therefore, each run of the simulation model with the same parameters gives slightly different results. In my thesis, I will try to design the GA so that it can deal with this type of randomness and provide near optimal solutions.

The second application is a gene clustering problem. The clustering problem is a search for natural groupings in data to simplify the description of a large set of multivariate data. With the availability of a nearly complete sequence of human genome, a wealth of DNA sequence data has been produced. The stage now has been set for the next task, which is the identification of the biological function of these genes in the sequences. Only this knowledge will enable researchers to establish correspondence between genes and their functions in the cell, leading to therapeutic discoveries. A major difficulty is that the details of the phenomena taking place within cells are not fully understood. Identification of the phenomena and the genes involved and determining how the genes interact constitute a major challenge in genomics.
One approach to meeting this challenge is to identify groups of co-expressed genes. Clustering gene expression data into groups reduces the unmanageable volume of data into datasets that can be more easily handled by biologists. In our GA application, our goal is to determine the centroids of gene clusters in order to identify possible associations between the genes and their functions in the cells.

In this approach, each gene is represented by an expression profile (a series of numerical values of its activities at different time points). We try to cluster genes in order to decide which genes exhibit similar behavior and understand the cause of a phenomenon. We assume that the number of clusters in this problem is given by biologists.

If we have \( n \) numerical values for each gene in its expression profile, the clustering problem will be \( n \) dimensional. Every group has a centroid in this space. Positioning these centroids is our decision problem. Then each gene is assigned to the closest centroid. To get the optimal solution, we need to find a set of clusters with the minimum total distance between genes and their groups’ centroids.

We use the continuous variable genetic algorithm to find the optimal solution for both of these applications. Decision variables of the queuing network design problem and the coordinates of centroids are continuous. The gene clustering problem is more complex than a queuing network because gene clustering has more locally optimal solutions. The GA may get stuck in one of these locally optimal solutions instead of finding the globally optimal solution.
Chapter 2

Evolutionary Computation and Genetic Algorithm

The invention of electronic computers has been one of the most revolutionary developments in the history of science and technology. Since the beginning of the computer age, one of the aims of computer scientists has been developing artificial intelligence. Early computer scientists were motivated by the idea of having computer programs with intelligence, which allows computers to learn and to control their environments. In addition to their interest in electronics, these pioneers were also interested in biology and psychology. They looked to natural systems for guiding examples. From the earliest days computers were applied to modeling the human brain. Some of these activities have increased and gained popularity in the early 1980s. The first area of growth was in the field of neural networks, and the second was in the field of what is now called the evolutionary computation. Genetic algorithms are among the best examples of evolutionary computation (Mitchell 1).
2.1 A Brief History of Evolutionary Computation

In the 1950s and 1960s, several computer scientists independently studied evolutionary systems. They believed that evolution could be used as an optimization tool for solving engineering problems. The idea was to start with a random set of initial population of candidate solutions for a given problem and evolve this population using operators inspired by biological process of evolution.

In the 1960s, Rechenberg (1965, 1973) introduced a method called evolution strategies, which he used to optimize real parameters for devices such as airfoils. This method was further developed by Schwefel (1975, 1977). The field of evolution strategies has remained an active area of research, mostly developing independently from the field of genetic algorithms. Fogel, Owens, and Walsh (1966) developed another technique called evolutionary programming, in which candidate solutions for a given problem were represented as finite-state machines. These finite-state machines were evolved by randomly mutating their state-transition diagrams and selecting the fittest solution. Evolution strategies, evolutionary programming, and genetic algorithms together form the basis of the field of evolutionary computation.

John Holland, motivated by the study of adaptation, invented genetic algorithms (GAs) in the 1960s. GAs were subsequently developed by Holland and his students and colleagues at the University of Michigan. In contrast with evolution strategies and evolutionary programming, Holland's original goal was not to design algorithms to solve specific problems. He was interested in studying the phenomenon of adaptation and developing ways to integrate this idea into computer systems. Holland's GA is a method for moving from one population of "chromosomes" (e.g., binary strings) to a new population by using a kind of "natural selection" together with the genetics-inspired operators of crossover, mutation, and inversion. Each chromosome consists of "genes" (e.g., bits), with each gene being an instance of a particular "allele" (e.g., 0 or 1). The selection operator chooses those chromosomes in the population that will be allowed to reproduce, and generally the fitter chromosomes produce more offspring than the less fit ones. Crossover exchanges subparts of two chromosomes, mimicking biological
recombination between two single-chromosome ("haploid") organisms; mutation randomly changes the allele values of some locations in the chromosome; and inversion reverses the order of adjacent genes in the chromosome.

Population-based selection, combined with crossover, inversion, and mutation, was a major innovation introduced by Holland. He was also the first to try to develop a firm theoretical basis for evolutionary computation.

Lately, interaction among researchers studying evolutionary computation methods has increased and the various methods that we have mentioned above are now described by the term general term "genetic algorithm" (Mitchell 2-3).

2.2 Biological Terminology

All known living organisms consist of cells, and each cell contains the same set of one or more chromosomes (strings of DNA) that serve as a blueprint for the organism. A chromosome can be conceptually divided into genes (functional blocks of DNA), each of which encodes a particular protein. One can think of a gene as encoding a trait, such as eye color. The different possible settings for a trait (e.g., blue, brown, and hazel) are called alleles. Each gene is located at a particular locus (position) on the chromosome.

In genetic algorithms, the term chromosome typically refers to a candidate solution for a problem, often encoded as a bit string. Genes are either single bits or short blocks of bits that encode a particular element in the candidate solution. Crossover typically consists of exchanging genetic material between two single-chromosome haploid parents. Mutation consists of complementing the bit value at a randomly chosen locus (or, for larger alphabets, replacing a symbol at a randomly chosen locus with a randomly selected symbol). The fitness of an organism is measured by success of the organism in its life (i.e., its survival), while in GAs, the fitness of a solution determines its acceptability (Mitchell 5-6; Obitko).
2.3 Search Spaces

In computer science, it is common to search among a collection of candidate solutions for a desired solution to a problem. The term "search space" refers to some collection of these candidate solutions (Mitchell 6). For example, in the queuing network design problem, there are a lot of decision variables, such as service rates, routing probabilities, and buffer sizes. The GA is an efficient method to find an acceptable solution, even in highly multidimensional search spaces with complicated objective functions.

Another example is the gene clustering problem. Suppose we have gene expressions that have ‘n’ numerical values for each gene in its expression profile. These genes are clustered in ‘m’ groups. Every group has a centroid in this space. These centroids’ positions are the decision variables of the clustering problem. Each gene is assigned to the closest centroid. To get an optimal solution, we need to find a set of clusters with minimum total distance between genes and their clusters’ centroids. The search space for each of the m clusters is ‘n’ dimensional.

2.4 Genetic Algorithm

The GA is an iterative process that operates on a population, i.e., a set of candidate solutions. Each solution is obtained by means of an encoding/decoding mechanism, which enables us to represent the solution as a chromosome and vise versa. Initially, the population is randomly generated. Every individual in the population is assigned, by means of a fitness function, a fitness value that reflects its quality with respect to solving the particular problem. The algorithm uses this quantitative information to guide the search. The whole process is described in Figure 2.1.
In each cycle, we first determine the fitness of each candidate solution. The next stage is selection, where a temporary population is created in which the fittest individuals (those corresponding to the best solutions contained in the population) are likely to have a higher number of instances than less fit individuals. The reproductive operators (crossover and mutation) are applied to the individuals in this population yielding a new population. Finally, some or all of the individuals of the original population are replaced by the newly created individuals. Less fit individuals are gradually eliminated from the population. The whole process is repeated until a certain termination criterion is achieved, usually after completing a predetermined number of iterations or reaching steady state (Alba and Cotta).

The population size in some GA applications can be changed during program execution. In this study we preferred to use a fixed population size.
In the algorithm, we do not necessarily reject the new solutions that do not improve the existing ones, based on the fact that some new zones of the search space may lead to more fit solutions.

GAs are heuristics and thus they do not ensure an optimal solution. The behavior of these algorithms is stochastic, so they may potentially present different solutions in different runs of the same algorithm. For this reason, it is common to perform several independent runs when studying a particular problem, and retain the best solution.

2.4.1 Encoding

Each chromosome contains information about a solution that it represents. Encoding of the chromosomes depends mainly on the problem to be solved. It is the first question to ask when starting to solve a problem with GA. A variety of encoding methods have been tested and verified (Obitko).

2.4.1.1 Binary Encoding

Binary Encoding is the most famous GA encoding method. The first research of GA used this type of encoding, and it is simple to implement. In binary encoding, every chromosome is a string of bits whose values equal 0 or 1.

```
101100101100101011100101
111111100000110000011111
```

**Figure 2.2**: Binary Encoding Example

2.4.1.2 Permutation Encoding

Permutation encoding can be used in ordering problems, such as the traveling salesman problem or the task ordering problem. Every chromosome is a string of ordered entities. The entities may be anything such as numbers, city names, tasks, etc. All of the chromosomes have the same entities but in a different order. In permutation encoding, a GA is used to find the optimal order of these entities.
2.4.1.3 Value Encoding

Direct value encoding can be used in problems where decision variables can take more complex values such as real numbers, characters, or any object(s). Every chromosome is a sequence of these values. For example, real value encoding is used in finding weights for a neural network.

<table>
<thead>
<tr>
<th>1.2324</th>
<th>5.3243</th>
<th>0.4556</th>
<th>2.3293</th>
<th>2.4545</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABDJEIFJDHDIERJFDLDFLFEGT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(back), (back), (right), (forward), (left)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.4: Examples of Different Types of Value Encoding

2.4.1.4 Tree Encoding

In tree encoding, every chromosome is represented as a tree of objects, such as functions or operators in a programming language. Tree encoding is suitable for any structure that can be encoded by trees.

Figure 2.5: Examples of Different Types of Tree Encoding

<table>
<thead>
<tr>
<th>+</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>/</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

( + x ( / 5 y ) )

<table>
<thead>
<tr>
<th>do until</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
</tr>
<tr>
<td>wall</td>
</tr>
</tbody>
</table>

( do_until step wall )
2.4.2 Evaluation

A chromosome is evaluated by a fitness function to determine the quality of the solution, i.e., how effective it is in solving the problem (Hsiung and NeuroDimension). There are many different kinds of fitness functions. The most appropriate fitness function depends on the application domain. Fitness functions range from a simple formula to a complex simulation program. The input of the fitness function is the chromosome and the output is the fitness value of this chromosome.

2.4.3 Selection

Selection is the process by which chromosomes are chosen, according to their fitness values, to be used as parents for the next generation. Copying chromosomes according to their fitness values means that chromosomes with higher values have a higher probability of contributing offspring to the next generation. Selection may be implemented in a number of ways such as Roulette Wheel, Rank-Based Selection, and Steady State Selection (Back 165-166, Goldberg, Obitko, and Pohlheim).

2.4.3.1 Roulette Wheel Selection

In Roulette Wheel Selection, chromosomes are selected as parents according to their fitness. The probability that a given chromosome will be selected is its fitness value divided by the sum of the fitness values for the entire population. Chromosomes with high fitness values have more chance to be a parent.

Imagine a roulette wheel where all the chromosomes in the population are placed. The size of the chromosome’s section in the roulette wheel is relative to the fitness value of that chromosome. Clearly, the chromosomes with higher fitness value are more likely to be selected.

**Table 2.1**: Fitness Value and Selection Probability for Roulette Wheel Selection

<table>
<thead>
<tr>
<th>Chromosomes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness value</td>
<td>2.0</td>
<td>1.8</td>
<td>1.6</td>
<td>1.4</td>
<td>1.2</td>
<td>1.0</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
<td>0.0</td>
</tr>
<tr>
<td>Selection probability</td>
<td>0.18</td>
<td>0.16</td>
<td>0.15</td>
<td>0.13</td>
<td>0.11</td>
<td>0.09</td>
<td>0.07</td>
<td>0.06</td>
<td>0.03</td>
<td>0.02</td>
<td>0.0</td>
</tr>
</tbody>
</table>
2.4.3.2 Rank-Based Selection

In Rank-Based Selection, the population is sorted according to fitness values. Rank-Based Selection is similar to the Roulette wheel selection, except that the probability for selection depends upon the rank of the fitness value within the sorted sequence. The chromosome that has the smallest fitness value will have the rank 1, the second worst 2 etc. and the best will have the rank N (number of chromosomes in population). The probability for chromosome to be selected is its rank (1…N) divided by the sum of the ranks for the whole population.

**Table 2.2: Fitness Value and Selection Probability for Rank-Based Selection**

<table>
<thead>
<tr>
<th>Chromosomes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness value</td>
<td>0.1</td>
<td>0.2</td>
<td>1.6</td>
<td>2.4</td>
<td>4.1</td>
<td>6.5</td>
<td>7.9</td>
<td>10</td>
<td>11</td>
<td>190</td>
</tr>
<tr>
<td>Selection probability</td>
<td>0.018</td>
<td>0.036</td>
<td>0.055</td>
<td>0.073</td>
<td>0.091</td>
<td>0.109</td>
<td>0.127</td>
<td>0.145</td>
<td>0.164</td>
<td>0.182</td>
</tr>
</tbody>
</table>

Rank-based selection is a better choice than roulette wheel selection for chromosomes that have large differences between their fitness values. For Table 2.2, if we try to apply roulette wheel selection, the chromosome with fitness value 190 will have a probability 81% to be selected as a parent and a probability .81 + .81 * (1-.81) = 96% to be one of the two parents. Therefore, the next generation will be dominated by the genes from one chromosome.

2.4.3.3 Steady State Selection

In Steady State Selection, the population is sorted according to their fitness values. Good chromosomes (e.g., the top 50%) are selected as parents, with the same probability for creating new offspring, while other chromosomes with lower fitness values do not qualify to be parents.

**Table 2.3: Fitness Value and Selection Probability for Steady State Selection with 50%**

<table>
<thead>
<tr>
<th>Chromosomes</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness value</td>
<td>2.0</td>
<td>1.8</td>
<td>1.6</td>
<td>1.4</td>
<td>1.2</td>
<td>1.0</td>
<td>0.8</td>
<td>0.6</td>
<td>0.4</td>
<td>0.2</td>
</tr>
<tr>
<td>Selection probability</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
2.4.4 Crossover

Two chromosomes are selected to be parents to create two offspring. Each offspring is recombination of genetic material from each of the two parents. Crossover may be implemented in a number of ways such as Single Point, Two Point, Uniform, and Arithmetic Crossover (Hsiung, NeuroDimension, and Obitko).

2.4.4.1 Single Point Crossover

Single point crossover operates by picking a random point within the two parent chromosomes, and then exchanging the genes of the two chromosomes at or below this point to produce two new offspring.

![Single Point Crossover](image)

*Figure 2.6: Single Point Crossover*

2.4.4.2 Two Point Crossover

Two point crossover operates by picking two random points within two parent chromosomes, and then exchanging the genes between these points in each parent to produce two new offspring.

![Two Point Crossover](image)

*Figure 2.7: Two Points Crossover*

2.4.4.3 Uniform Crossover

Genes are randomly copied from the two parents to compose two new offspring.

![Uniform Crossover](image)

*Figure 2.8: Uniform Crossover*
2.4.4.4 Arithmetic Crossover

An arithmetic operation is performed to make a new offspring.

![Figure 2.9: Arithmetic Crossover](image)

2.4.5 Mutation

Mutation takes place after crossover is performed. Mutation randomly changes a selected gene in the selected offspring. This occurs at an assigned rate, usually very infrequently. The purpose of mutation is to prevent falling into a locally optimal solution of the solved problem (Obitko). Mutation for continuous GA may be implemented in a number of ways such as Value, Micro, and Add Value Mutation.

2.4.5.1 Value Mutation

In continuous GA, each chromosome consists of many genes. Each gene is represented by a sequence of digits. Value mutation occurs by setting a whole gene from the selected offspring chromosome to a new random value.

2.4.5.2 Micro Mutation

This mutation occurs by randomly changing the value of a digit (a part of a gene) from the selected offspring chromosome.

2.4.5.3 Add Value Mutation

This mutation occurs by adding or subtracting a small random number to or from the value of a gene from the offspring chromosome.
2.4.6 Parameters of GA

Several different parameters control the behavior of GA, including Crossover Probability, Mutation Probability, Population Size, and Elitism Number.

2.4.6.1 Crossover Probability

Crossover probability is the probability that crossover will occur between selected chromosomes during the creation of the next generation. If there is no crossover, offspring will be exact copies of their parents. If there is a crossover, offspring are made from parts of their parents' chromosomes. Crossover is performed in the hope that new chromosomes will exhibit better fitness than the parent chromosomes.

2.4.6.2 Mutation Probability

Mutation probability is the likelihood of mutation taking place in chromosomes selected during the creation of the next generation. If there is no mutation, offspring are taken after crossover (or copy) without any change. If mutation is performed, part of the chromosome is changed. Mutation is made to prevent GA from stopping at a local optimal solution. Mutation should not occur very often, because GA will tend to become a random search.

2.4.6.3 Population size

Population size is the number of chromosomes in a population. If there are too few chromosomes, GAs will have fewer possibilities to perform crossover and only a small part of search space will be explored. On the other hand, if there are too many chromosomes, then GAs will slow down.

2.4.6.4 Elitism Number

Elitism is the name of the method that first copies the best chromosome (or a few of the best chromosomes) to the new population without crossover and mutation (Obitko). The rest of the population is constructed by using GA operations (selection, crossover, and
Elitism can rapidly increase the performance of GA, because it prevents the loss of the best chromosomes from the previous generation.

### 2.4.7 Random Number Generator

A random number generator is used by a GA to generate the candidate solutions for the initial population, to select the parents, to select the method and the location of crossover, and to select the method and value of the mutation. Most computer languages have a function that generates a pseudo-random number. A particular sequence of random numbers can be recreated by using the same random number seed. The goal of any random number generator is to produce a sequence of numbers that simulates the ideal properties of uniform distribution and independence as closely as possible.

The most widely used technique for generating random numbers is *Linear Congruential Method*. This technique that proposed by Lehmer [1951], produces a sequence of integers between zero and \((m - 1)\) according to the following recursive formula:

\[
X_{i+1} = (a X_i + c) \mod m, \quad i = 0, 1, 2, \ldots
\]  

(2.1)

The initial value ‘\(X_0\)’ is named the seed, ‘\(a\)’ is named constant multiplier, ‘\(c\)’ is the increment, and ‘\(m\)’ is the modulus. The selection of these values affects the statistical properties and cycle length. These values has been widely tested and selected by Learmonth and Lewis, 1973; Lewis et al., 1969, to ensure that the characteristics preferred in a generator are most likely to be reached. The values, which selected are \(a = 7^5 = 16,807\), \(m = 2^{31} - 1 = 2,147,483,647\) (a prime number) and \(c = 0\), while the value of ‘\(X_0\)’ is changed by the user to control the repeatability of the stream. The cycle length for this generator with these selections is \(m - 1\), over than 2 billion (Banks 255-262).
Chapter 3

Genetic Algorithm Applications to Two Problems

The GA approach has proven to be suitable for many of the hardest computational problems in a variety of fields, and now appears to be the method of choice for application problems in fields ranging from bioinformatics to antenna design. (A search by Google for the phrase “genetic algorithm 2003” yields 205,000 hits, while “genetic algorithm” yields about 775,000 hits.) Many computational problems, including operations, design, and planning, require searching for a solution in a very large space. For example, a problem of protein engineering requires a search through a vast number of possible amino acid sequences for a protein with specified properties. This problem is very complex since a protein is usually formed of 10–100,000 amino acids and there are 20 different types of amino acids (Paustian).

In another example of finding the optimal shape of aircraft wings, the GA was also successfully employed. In his doctoral thesis, Akira Oyama used an improved version of continuous GA, Adaptive Range Genetic Algorithms (ARGAs), to find the
optimal parameters for aircraft wings. While the problem was very complicated, he achieved good results using ARGAs. This problem is usually parameterized by more than a hundred design parameters involved in an extremely complicated fitness function. The result is considered to be the global optimum, and thus ensures the feasibility of the real-coded ARGAs in aerodynamic designs (Oyama).

The goal of this study is to investigate the efficacy of the GA for continuous variable optimization in two complex problem areas. The first problem is a computer (queuing) network problem in which we are supposed to determine the optimal settings of the speed parameters. The major difficulty of this problem is that each evaluation of a solution requires running a simulation program that provides a random performance measure. The second problem is related to the analysis of gene expression data for which the solution needs to cluster thousands of genes into a small number of groups that work together. We will explain these problems and their GA solutions in a greater detail in the following sections.

3.1 Queuing Network Problem

Designing early computer and communications systems did not require mathematical optimization techniques. The systems at that time were not so complicated and the numbers of possible design choices were limited. This situation has changed dramatically for modern systems. These systems have become very complex and require high performance. Therefore, an optimal design is not easily observable and the number of candidate designs can be enormous.

The design problem may include cost, performance, dependability, and functional specifications. The cost specifications may include a minimization of the total cost, or may provide a constraint in the optimization model. The performance specification states how well the specified functions should perform in terms of throughput\(^1\), response time,

---

\(^1\) Throughput is the mean number of jobs whose processing is completed in a single unit of time.
The dependability specification states how the system tolerates component faults. This specification also includes minimum acceptable reliability, availability, etc.

The system design methodology takes the design specification as an input and converts it into a system configuration that fulfills the specification. The system configuration is characterized by a set of hardware parameters such as the number of processors, the size of memories, the speed and capacity of resources, and other architectural parameters, including the system topology. It is very difficult task to choose the configuration that best meets the system specification.

To formulate the optimization problem, it is important to determine the system parameters that can be varied and hence considered to be decision variables. Below are some objective examples:

- Maximization of throughput with cost constraint.
- Minimization of cost for given throughput.
- Minimization of the mean response time with cost constraint (Bolch 557-558).

### 3.1.1 Problem description

A challenging network problem (Bolch 565 problem 11.1) has been borrowed for this study. The aim of this problem is to maximize the throughput (the number of jobs that pass through a point in a single unit of time) with a cost constraint in a closed queuing network.

This closed network is a multiprogramming computer system with a CPU and three disk devices as shown in Figure 3.1. The CPU is named as node 1 and the three disks are named as nodes 2, 3, and 4. In our study, the degree of multiprogramming, $K$ (the number of jobs in the systems) is fixed at a value of four. Since the network is closed, no new jobs may enter the network and none of the current jobs may leave the network.
The corresponding routing probabilities are \( p_{11} = 0.05, p_{12} = 0.5, p_{13} = 0.3, p_{14} = 0.15, p_{21} = p_{31} = p_{41} = 1.0 \). That is, after a job gets out of the CPU (node 1), there is 5% probability for it to return back to the CPU, 50% probability to go to node 2 which is the first disk and so on. After a job gets out of any disk, it goes to the CPU.

The service rate \( \mu_i \) is the average number of jobs that can be completed in a node \( i \) in a single unit of time. In this problem, service rates are the values we need to determine. We use an exponential distribution with an average \( 1/\mu_i \) to calculate the service time for a job to be completed in node \( i \). Each time a job enters a node, we use the following formula to calculate the service time \( t_i \) for this job:

\[
 t_i = -\frac{\ln(\text{rand})}{\mu_i} \tag{3.1}
\]

where \text{rand} is a random number in (0,1) and \( \ln \) is the natural logarithm.

The node is either idle or busy serving a job. If a job reaches an idle node, the node will immediately start serving this job. Once the service is completed, the job will move on to the next node. If a job reaches a busy node, the job joins the waiting queue. Each node has its own waiting queue. When the node finishes serving all the items in the queue, it returns back to the idle state.

The throughput is the number of jobs passed through a point in a single unit of time. Throughput measures the performance of the network: the higher the throughput, the better the network. In this problem, throughput is measured on the path from the output of the CPU to the input of the CPU as shown in Figure 3.1.
3.1.2 Cost Functions

The simplest cost function is the linear function. For a resource that has a service rate\(^2\) \(\mu_i\) as a decision variable, the linear cost function will be \(c_i \mu_i\), where \(c_i\) is the cost factor for service rate \(\mu_i\) of resource \(i\). The total cost is given by:

\[
COST = \sum_{i=0}^{N} c_i \mu_i
\]  

(3.2)

The linear cost function does not cover all the possibilities for the cost function. The more realistic estimates are obtained by using a nonlinear cost function:

\[
COST = \sum_{i=0}^{N} c_i \mu_i^{\alpha_i} \quad \text{where} \quad \alpha_i > 1
\]  

(3.3)

The service rate \(\mu_i\) depends on the speed \(v_i\) of device \(i\) and the mean number of work units \(d_i\) per visit:

\[
\mu_i = \frac{v_i}{d_i}
\]  

(3.4)

---

\(^2\) Service rate is the mean number of jobs whose processing is completed in a node \(i\) in a single unit of time.
For a CPU, the mean number of instructions per CPU burst is given by $d_i$ and the CPU speed (the number of instructions per time unit) by $v_i$. For a storage device, $d_i$ is the mean number of words in an I/O operation and $v_i$ is the speed of the storage device in words per time unit (Trivedi).

In the case of a multiprogramming computer system, there is an additional linear cost which is the cost for the main memory $C(K)$. This cost depends on the degree of multiprogramming $K$ (the number of jobs in the systems). The linear memory cost $C(k)$ is given by the following formula:

$$C(K) = C_m K$$

(3.5)

The total cost function is then given by:

$$COST = C_m K + \sum_{i=0}^{N} C_i \mu_i^a$$

(3.6)

(Bolch 558-559).

3.1.3 Maximization of the Throughput Based on BFS

This analytical method was introduced by Bolch, Flesischmann and Scherppel (Bolch 560-561). It is called BFS for short. It is an approximation method, so the results are usually close to the optimal values. BFS is demonstrated using a closed queuing network example with $N$ stations and $K$ jobs. The “optimal” values are obtained by solving these equations.

$$\lambda^* = \frac{COST \cdot K}{\left(\sum_{i=1}^{N} \sqrt{c_i e_i}\right)^2 + K \sum_{i=1}^{N} c_i e_i}$$

(3.7)

$$\mu_i^* = \lambda^* \cdot e_i \left(\sum_{j=1}^{N} c_j e_j \right) + 1$$

(3.8)
where $\lambda^*$ is the optimal throughput.  

$K$ is the number of jobs in the system, which is four in this problem.  

$e_i$ is the visit ratio for node $i$.  

All the nodes service one job at a time.  

For Figure 3.1, the cost assumed factors $c_i$ and exponents $\alpha_i$ are given in Table 3.1.  

**Table 3.1:** The Cost Factors $c_i$ and Exponents $\alpha_i$ for Queuing Network Components  

<table>
<thead>
<tr>
<th>Component</th>
<th>$c_i$</th>
<th>$\alpha_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>131.9</td>
<td>0.55</td>
</tr>
<tr>
<td>Disk1</td>
<td>11.5</td>
<td>1.00</td>
</tr>
<tr>
<td>Disk2</td>
<td>54.2</td>
<td>0.67</td>
</tr>
<tr>
<td>Disk3</td>
<td>54.2</td>
<td>0.67</td>
</tr>
</tbody>
</table>

The total budget COST for this problem is assumed to be 500 with $C_m = 50$ for equation (3.6). The optimal result for this problem after applying the BFS method is given in Table 3.2.  

**Table 3.2:** The Optimal Result after Applying BFS  

<table>
<thead>
<tr>
<th>$K$</th>
<th>$\lambda$</th>
<th>$\mu_1$</th>
<th>$\mu_2$</th>
<th>$\mu_3$</th>
<th>$\mu_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.076</td>
<td>1.967</td>
<td>1.979</td>
<td>0.847</td>
<td>0.571</td>
</tr>
</tbody>
</table>

### 3.1.4 GA Formulation  

The BFS method gives the optimal solution to the above problem only under the assumptions of some specific configuration and service time distributions. In BFS, the time distributions of service times are exponential. Here, we will compare how the GA for continuous variables will perform against the above known optimal settings of the parameters. Every solution provided by the GA needs to be evaluated in the simulation model. The simulation model is very flexible in that we can use any kind of time distribution. To compare the result of GA with BFS method, we are going to use the exponential distribution in the simulation model. The results will shed light on the appropriateness of the approach for other analytically intractable versions of this and other problems.
In the above problem, the decision variables are the speeds of the processor and the disks, which will maximize the fitness value expressed as a composite function of the throughput and the cost. The cost is introduced into the objective function with a penalty coefficient using the same concept of the penalty functions that leads to an analytical solution in the BFS method. Accordingly, if the cost exceeds the cost limit of 500 (Bolch 565 problem 11.1), the fitness value is penalized proportional to the violation amount; otherwise, the fitness value is equal to the throughput. That is,

\[
F(\mu_1, \mu_2, \mu_3, \mu_4) = \lambda - \eta(c),
\]

where \( \lambda \) is the throughput obtained by the simulation; 
\( c \) is the cost of the decisions \( \mu_1, \mu_2, \mu_3, \) and \( \mu_4 \); 
\( l \) is the cost limit which is 500; and

\[
\eta(c) = \begin{cases} 
0, & \text{if } c = l \\
0.02(c - l), & \text{if } c > l 
\end{cases}
\]

The value for the cost penalty factor, 0.02, is set by a simple initial trial and error method, so that if the constraint violation occurred, it would remain close to the limit value, 500.

In order to obtain a fitness value for a particular setting of the parameters, a simulation program has to be run for this queuing network to estimate the throughput in a short period. The simulation program is constructed with the Queuing-Network class, which extends Evaluation class (see appendix B). The input for this class is the service rates and the output is the random throughput. The simulation program uses a random exponential distribution to generate service times for the components of the network. Therefore, every simulation run with the same parameters does not necessarily give the same results.

3.1.4.1 Simulation Program

A queuing network simulation program is constructed in Java. This program is flexible enough to design any queuing network configuration. It employs 25 classes altogether. The input parameters of the simulation program are the speed of the processor and disks
and the output is the estimated throughput. In order to test the correctness of this simulation program, it was executed with the optimal service rates as input to see if the average throughput is close to the analytical result obtained by the BFS approach. The simulation program ‘QueuingNetworkCheckBFSSolution.java’, of the appendix B, was run for 100,000 time units and the average throughput was collected at every 10,000 time units. The results are shown in Table 3.3.

Table 3.3: The Average Throughput for Queuing Network with the Optimal Parameter Settings of the BFS

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_i$</td>
<td>0.0742</td>
<td>0.0810</td>
<td>0.0747</td>
<td>0.0798</td>
<td>0.0752</td>
<td>0.0779</td>
<td>0.0763</td>
<td>0.0755</td>
<td>0.0804</td>
<td>0.0762</td>
</tr>
</tbody>
</table>

From Table 3.3, we obtained the following: the overall average = 0.07712; $S^2 = 6.2E-06$; $t_{0.05/2.9} = 2.262159$, the half width hw = 0.001781 with the 95% confidence interval; lo = 0.075339; up = 0.078901. The 95% confidence level the throughput is in 0.07712+/-0.001781. Thus, the BFS result 0.076 fills inside this range.

3.1.4.2 Solving the Problem by GA

Our approach consists of the following three steps:

**Step 1: Running GA Component**

The first step is to run the standard GA. In the problem, the optimal values sought are the speed of the processor and drivers and the fitness value as a composite function of the throughput and the cost constraint with the penalty. GA algorithm was run with the following parameters, which were determined by the trial and error to obtain the best results:

- Population size = 500
- Selection method = Steady State Selection with 25% selection rate
- Crossover method 1 = Simple point crossover with 37.5% crossover rate
- Crossover method 2 = Micro crossover with 37.5% crossover rate
- Mutation method 1 = Value Mutation with 2.5% mutation rate
Mutation method 2 = Add Value Mutation with 2.5% mutation rate
Termination Criterion = Best Value Didn’t Change For 50 Generation
Elitism Number = 1

Each chromosome contains information about a solution that it represents. A solution consists of the processor speed and the speeds of the three drivers. A chromosome needs to encode these four values. Value encoding is chosen as the best method of encoding for this problem. The length of each variable is 5 digits and the total length of a chromosome is 5 * 4 = 20 digits. The first five digits are converted into the processor speed, the second five digits are converted into the speed of the first driver, and so on.

The speed range for all the four parameters is restricted in the range [0.1, 5). To convert the five digit number into a speed, the following formula is used:

\[ \text{Speed} = 0.1 + (5 - 0.1) \times \frac{\text{five digit number}}{100,000} \]  

(3.10)

The four speeds are inputs for the simulation program and the output is the throughput. The run time of the simulation program for each chromosome is limited to a short consecutive period of 1000 time units to obtain a quick evaluation. This process is repeated five hundred times for every candidate solution the population for each generation.

**Step 2: Check the Best Five Chromosomes of GA:**
In this step, the best five chromosomes are reevaluated for long period of time (10,000 time unit) in order to obtain a better estimate of their performance. The chromosome with the best (largest) fitness value is selected for next step.

**Step 3: Run for a Long Time:**
In this step, we use the parameters of the best chromosome as input for the simulation program and run it for a long period of time. The new period is 100,000 time units. The average estimate of throughput obtained at every 10,000 time units.
3.1.4.3 Confidence Interval built on Simulation Output

At this point, we need to give some statistical background. Simulation is a random process. Each run with the same parameters returns different results. To estimate the average results with a certain level of confidence for the simulation output, a t-distribution is used. The t-distribution is a theoretical probability distribution. It is a bell-shaped curve, similar to the standard normal curve. In fact, it approaches the normal distribution as the degree of freedom gets larger. It differs from the standard normal distribution in that it has an additional parameter, called degrees of freedom (df), that changes its shape, as seen in Figure 3.2(Stockburger).

Degrees of freedom, usually symbolized by df, is a parameter of the t-distribution that can be any number greater than zero. Setting the value of df defines a particular member of the family of t-distributions. The smaller the df, the flatter the shape of the distribution, resulting in greater area in the tails of the distribution.

As the df increase, the t-distribution approaches the standard normal distribution. The standard normal curve is a special case of the t-distribution when df = ∞. T-distribution approaches the standard normal distribution relatively quickly, such that when df=30, the two are almost identical. In simulation, the degree of freedom equals the number of sampling point minus one. In our case, we make 10 runs, and therefore df = 9.
Hence, we can calculate the half width of the interval where \((1-\alpha)\) percent of the samples are contained.

\[
hw = t_{\alpha/2, n-1} \sqrt{\frac{s^2}{n}}
\]  

(3.11)

where \(s^2\) is the variance;

\(n\) is the number of samples;

\(\alpha\) is the significance level or the probability associated with the two tailed Student’s t-distribution;

The values of \(t_{\alpha/2, n-1}\) are given in the t-distribution table.

### 3.1.4.4 Numerical Results

Execution of QueuingNetworkMainProgram.java (Appendix B) took seven minutes on a computer with an AMD 800 MHz CPU and 376MB RAM. The GA created 65 generations before reaching the optimal solution. The results are listed in Table 3.4:

<table>
<thead>
<tr>
<th>(i)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda_i)</td>
<td>0.0760</td>
<td>0.0725</td>
<td>0.0812</td>
<td>0.0818</td>
<td>0.0765</td>
<td>0.0792</td>
<td>0.0783</td>
<td>0.0781</td>
<td>0.0711</td>
<td>0.0793</td>
</tr>
</tbody>
</table>

From Table 3.4, we conclude the following: the overall average = 0.0774; \(S^2 = 1.2E-05\); \(t_{0.05/2,9} = 2.26\); \(hw = 0.002483\); \(lo = 0.074917\); \(up = 0.079883\). With 95% confidence level, the throughput is 0.0774\(\pm\)0.002483.

By comparing the BFS results and GA results, we find that the GA reaches a solution very close to the optimal as seen in Table 3.5 within the cost constraint.
### Table 3.5: Comparison of GA and BFS Results

<table>
<thead>
<tr>
<th></th>
<th>$\mu_1$</th>
<th>$\mu_2$</th>
<th>$\mu_3$</th>
<th>$\mu_4$</th>
<th>Total cost</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>1.967</td>
<td>1.979</td>
<td>0.847</td>
<td>0.571</td>
<td>499.84</td>
<td>0.076</td>
</tr>
<tr>
<td>GA</td>
<td>1.972</td>
<td>2.018</td>
<td>0.801</td>
<td>0.590</td>
<td>499.58</td>
<td>0.07740 +/-0.00248</td>
</tr>
</tbody>
</table>

By comparing GA with BFS method, we found that the GA gives very good results, close to the “optimal” solution found by the BFS method. But it does not require the complex mathematical analysis and the assumptions about specific configurations or service time distributions. The GA proves to be a good heuristic approach to the queuing network design problem.

### 3.2 Clustering Problem and Gene Expressions

The clustering problem of points multidimensional space is a combinatorial optimization problem where the goal is to partition a set of data objects into a predefined number of clusters in such a way that similar objects are grouped together while the dissimilar objects are put in separate groups (Kivijarvi). Organizing data into sensible groupings is one of the most fundamental modes of human understanding and learning. Clustering is used in many diverse fields, such as psychology, zoology, biology, sociology, artificial intelligence, information retrieval, and botany.

The difficulty of clustering stems from the large number of possible distinct partitions that exist in the search space. This number is formally stated as follows: given $N$ data points in a $D$ dimensional metric space, determine a partition of the patterns into $K$ clusters, such that the number of possible of distinct partitions will be (Jain 90-91):

$$S(N, K) = S(N - 1, K - 1) + K S(N - 1, K)$$

The boundary conditions on this equation are: $S(N, 1) = 1$, $S(N, N) = 1$, $S(N, K) = 0$ if $K > N$
The solution to the partial difference equation is called a Stirling number of second kind and is as follows:

\[
S(N, K) = \frac{1}{K!} \sum_{i=1}^{K} (-1)^{K-i} \binom{K}{i} i^N
\]  
(3.13)

For example, using the above formula, there are only 34,105 distinct partitions of 10 objects into four clusters, but this number explodes to approximately 11,259,666,000 if 19 objects are to be partitioned into four clusters. Clearly, the complete list of all possible partitions is not computationally feasible, even for a small number of patterns.

### 3.2.1 Gene Expressions

With the availability of a nearly complete sequence of human genomes, we have a wealth of DNA sequence data. The stage now has been set for the next task, which is identification of the biological function of these genes in the sequences. Only this knowledge will enable researchers to establish correspondence between genes and their functions in the cell, leading to therapeutic discoveries.

A major difficulty is that the details of the phenomena taking place within cells are not fully understood, and many will probably remain to be discovered. Identifying the phenomena and the genes involved and determining how the genes interact constitute a major challenge in genomics.

One approach to meeting this challenge is to identify groups of co-expressed genes. Clustering gene expression data into groups reduces the unmanageable volume of data into datasets that can be more easily handled by biologists. In our GA application, we will determine the centroids of gene clusters in order to find the association between the genes and their functions in the cells (Fogel 219-220).

In this approach, each gene is represented by an expression profile (a series of numerical values of its activities). We try to cluster genes in order to decide which genes
exhibit similar behavior and understand the cause of a phenomenon. The number of clusters in this problem is given by biologists.

### 3.2.2 K-mean Method to Clustering

K-means clustering generates a specific number of disjoint, flat (non-hierarchical) clusters (Predictive). This method is a frequently used approach for large cases and we will use it for comparison.

According to “K-means Clustering Overview” (Predictive), the following are the assumptions of the K-means algorithm:

- There are always K clusters.
- There is always at least one item in each cluster.
- The clusters are non-hierarchical and they do not overlap.
- Every member of a cluster is closer to its cluster’s centroid than to any other cluster’s centroid.

Initially, the dataset is partitioned into K clusters according to some ad hoc procedure (e.g., picking K random data points as the centroids and assigning points to the closest centroids, or randomly assigning the data points to clusters so that each cluster has roughly the same number of data points). The centroids are updated to the means of the points in their cluster. Then each point is clustered to the closest centroids. This procedure of updating the centroids and reassigning the points is repeated until no data point is moved from one cluster to another. At this point, the clusters are stable and the clustering process ends, as explained in Figure 3.3. K-means algorithm gives very good result only when the initial partitioning is close to the optimal solution (Jain 97).
Partitioned dataset is into $K$ clusters according to some adhoc procedure

DO

Centroids $\leftarrow$ means of the clusters

FOR each data point

FOR $i=0$ to $K$

$d_i \leftarrow$ distance($Centroid_i$, point)

END FOR

Move point to the cluster that has the smallest $d_i$

END FOR

WHILE at least one point moved to another cluster

RETURN Centroids

Figure 3.3: K-means Algorithm Skeleton

Figure 3.4 shows how the K-means clustering algorithm works on an example with points in a two dimensional space (Fogel 222).

Figure 3.4: An Example of K-means
3.2.3 GA Formulation of the Clustering Problem

In this study, the GA application to solve the clustering problem was done in four different ways. All the applications try to determine the centroids of the clusters, which are continuous values. The first application uses the standard GA on the entire data set. The other applications modify the solutions at each stage of the calculations, or use only a subset of the data points in order to improve the run time of the algorithm.

As an initial test problem, a desired number of data points with a fixed number of attributes were created by using Gaussian distributions around some preset centroids. Thus, this data can be used to check the four different applications of the GA algorithm. Test data contains 1,500 points in a 5 dimensional metric space. These points are generated to form 15 clusters with 100 points in each. The clusters of this data are designed to be well separated in order to make the comparisons of the applications easy. After verification of the algorithm on this data, the next step was to apply the algorithm on the real data of a large gene expression problem.

3.2.3.1 Decisions Variables

The algorithm tries to group $N$ points in a $D$ dimensional metric space into $K$ clusters. Once the centroids are decided, every cluster has a centroid in this space. Finding these centroids is our decision problem. Each point is assigned to the closest centroid. To get an optimal solution we need to find a set of $K$ cluster centroids with minimum total distance to their clustering points. We use our continuous genetic algorithm to solve the problem.

3.2.3.2 Evaluation of Clusters

One of the most important choices in the clustering method is the objective function for evaluating the quality of clustering. A commonly used objective criterion is to minimize the sum of squared distance of the points to their clusters’ centroids.
The distance between a point and its cluster’s centroid is measure by Euclidean distance of the two feature vectors, which is calculated as (Jain 15):

\[
d(p, c) = \sqrt{\sum_{d=1}^{D} (p_d - c_d)^2}
\]

(3.14)

where \( p \) is a point and \( c \) is a centroid in \( D \) dimensional space.

The fitness value for a GA chromosome is the mean squared error, which is the total distance between the patterns and their groups’ centroids (the closest centroid to this pattern):

\[
\text{Fitness value} = \frac{1}{ND} \sum_{i=1}^{N} d(p_i, c_p)^2
\]

(3.15)

where \( N \) is number of data points.

3.2.4 Generation of test Data

Test data contains 1500 points in a 5 dimensional unit metric space. These points are generated randomly into 15 clusters with 100 points in each. First, the centroids of the clusters need to be located. These centroids were randomly created with the condition that the distance between any pair be at least 0.79. This number is set by a trial and error experimentation. In this five-dimensional space, the diagonal distance is:

\[
\sqrt{1^2 + 1^2 + 1^2 + 1^2 + 1^2} = 2.236
\]

(3.16)

To generate the points around the centroids, a Gaussian distribution is used with a standard deviation chosen randomly from a range [0.02, 0.04]. Using this procedure, 100 points are generated within the unit metric space for each of the centroids. If the generated point happened to lie outside the unit metric space, then it was ignored and a new one was generated until it was located within the unit space. Figure 3.5 shows the centers of 1500 points in a 2-dimentional projection.
3.2.5 Running GA with the Clustering Test Data

Four applications were applied on the test data. The purpose of these applications was to find the most effective GA in a large-scale clustering problem. These applications made different injects on the results. Each application includes some enhancement of the previous case.

3.2.5.1 Approach 1: Standard GA

In the first application, standard GA was applied without any adaptation. The GA was run with the following parameters, which were determined by the trial and error to obtain the best results:

Population size = 100
Selection method = Steady State Selection with 25% selection rate  
Crossover method = Simple point crossover with 65% crossover rate  
Mutation method = Value Mutation with 2% mutation rate  
Termination Criterion = Best Value Didn’t Changed For 50 Generation  
Number of runs = 10 with different initial generations.

**Table 3.6: Clustering Results after Applying GA**

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Generation</th>
<th>The Fitness of solution</th>
<th>Fitness of Solution Divided by Fitness of actual Centroids</th>
<th>% of Correctly Assigned Points</th>
<th>Mean Squared Error between Calculated and actual Centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98</td>
<td>1182</td>
<td>0.006341</td>
<td>1.85</td>
<td>90.6%</td>
<td>0.00419</td>
</tr>
<tr>
<td>2</td>
<td>55</td>
<td>1101</td>
<td>0.006342</td>
<td>1.85</td>
<td>90.6%</td>
<td>0.00587</td>
</tr>
<tr>
<td>3</td>
<td>73</td>
<td>1455</td>
<td>0.003799</td>
<td>1.11</td>
<td>100.0%</td>
<td>0.00011</td>
</tr>
<tr>
<td>4</td>
<td>70</td>
<td>1384</td>
<td>0.003641</td>
<td>1.06</td>
<td>100.0%</td>
<td>0.00010</td>
</tr>
<tr>
<td>5</td>
<td>76</td>
<td>1515</td>
<td>0.003535</td>
<td>1.03</td>
<td>100.0%</td>
<td>0.00010</td>
</tr>
<tr>
<td>6</td>
<td>52</td>
<td>1039</td>
<td>0.003807</td>
<td>1.11</td>
<td>100.0%</td>
<td>0.00013</td>
</tr>
<tr>
<td>7</td>
<td>64</td>
<td>1283</td>
<td>0.003829</td>
<td>1.12</td>
<td>100.0%</td>
<td>0.00015</td>
</tr>
<tr>
<td>8</td>
<td>90</td>
<td>1467</td>
<td>0.006721</td>
<td>1.96</td>
<td>90.8%</td>
<td>0.00299</td>
</tr>
<tr>
<td>9</td>
<td>72</td>
<td>1268</td>
<td>0.003748</td>
<td>1.09</td>
<td>100.0%</td>
<td>0.00012</td>
</tr>
<tr>
<td>10</td>
<td>87</td>
<td>1489</td>
<td>0.006609</td>
<td>1.93</td>
<td>90.4%</td>
<td>0.00435</td>
</tr>
<tr>
<td>Average</td>
<td>74</td>
<td>1318</td>
<td>0.004837</td>
<td>1.41</td>
<td>96.2%</td>
<td>0.00181</td>
</tr>
<tr>
<td>STDEV</td>
<td>15</td>
<td>170</td>
<td>0.001441</td>
<td>0.42</td>
<td>4.9%</td>
<td>0.00229</td>
</tr>
<tr>
<td>Max</td>
<td>98</td>
<td>1515</td>
<td>0.006721</td>
<td>1.96</td>
<td>100.0%</td>
<td>0.00587</td>
</tr>
<tr>
<td>Min</td>
<td>52</td>
<td>1039</td>
<td>0.003535</td>
<td>1.03</td>
<td>90.4%</td>
<td>0.00010</td>
</tr>
</tbody>
</table>

The main class for this run is the `ClusteringMainProgram1.java` in Appendix B.

Actual centroids represent the means of the Gaussian distributions, which were used to generate the points. The fitness value of the actual centroids for this data set is 0.003429.

### 3.2.5.2 Approach 2: Reordering Centroids before Crossing-over Operation

While running the GA algorithm to solve the clustering problem, each chromosome may have a different centroid order. That is, the naming of the clusters is not consistent from one solution to another. It will help a great deal in the convergence of the GA to find the associations among the cluster names and order them accordingly in the chromosomes.
before the crossovers. Figure 3.6 depicts the difficulty unless this association is made before the crossover between two chromosomes. In Figure 3.6.a, when the clusters C1 and C2 in parent 1 are forced to crossover with C2 and C1 in Parent 2, respectively, the offspring will be worse off than both of the parents. The fitness value of a solution stems from the positions of the centroids rather than this naming order.

**Figure 3.6: An Example for Crossover without/with Centroid Reordering**

Instead, after establishing the associated naming of the centroids in the parents based on the closeness of the centroids, we crossover the parents, as shown in Figure 3.6.b.

We modified the GA algorithm for clustering with the ordering of the centroids as described above and ran the program (*ClusteringMainProgram2.java* in Appendix B) on the same problem described in the previous section. Table 3.7 shows the new results.
Table 3.7: Clustering Results after Reordering Centroids in the GA.

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Generation</th>
<th>The Fitness of solution</th>
<th>Fitness of Solution Divided by Fitness of actual Centroids</th>
<th>% of Correctly Assigned Points</th>
<th>Mean Squared Error between Calculated and actual Centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>55</td>
<td>1092</td>
<td>0.00392</td>
<td>1.14</td>
<td>100%</td>
<td>0.00017</td>
</tr>
<tr>
<td>2</td>
<td>54</td>
<td>1060</td>
<td>0.00366</td>
<td>1.07</td>
<td>100%</td>
<td>0.00012</td>
</tr>
<tr>
<td>3</td>
<td>52</td>
<td>1035</td>
<td>0.00409</td>
<td>1.19</td>
<td>100%</td>
<td>0.00020</td>
</tr>
<tr>
<td>4</td>
<td>84</td>
<td>1695</td>
<td>0.00350</td>
<td>1.02</td>
<td>100%</td>
<td>0.00009</td>
</tr>
<tr>
<td>5</td>
<td>60</td>
<td>1181</td>
<td>0.00616</td>
<td>1.79</td>
<td>92%</td>
<td>0.00569</td>
</tr>
<tr>
<td>6</td>
<td>56</td>
<td>1105</td>
<td>0.00382</td>
<td>1.11</td>
<td>100%</td>
<td>0.00016</td>
</tr>
<tr>
<td>7</td>
<td>72</td>
<td>1420</td>
<td>0.00352</td>
<td>1.03</td>
<td>100%</td>
<td>0.00011</td>
</tr>
<tr>
<td>8</td>
<td>53</td>
<td>1042</td>
<td>0.00382</td>
<td>1.11</td>
<td>100%</td>
<td>0.00016</td>
</tr>
<tr>
<td>9</td>
<td>33</td>
<td>647</td>
<td>0.00730</td>
<td>2.13</td>
<td>90%</td>
<td>0.00323</td>
</tr>
<tr>
<td>10</td>
<td>56</td>
<td>1105</td>
<td>0.00360</td>
<td>1.05</td>
<td>100%</td>
<td>0.00010</td>
</tr>
<tr>
<td>Average</td>
<td>57</td>
<td>1138</td>
<td>0.00434</td>
<td>1.26</td>
<td>98.2%</td>
<td>0.00100</td>
</tr>
<tr>
<td>STDEV</td>
<td>13</td>
<td>271.4</td>
<td>0.00130</td>
<td>0.38</td>
<td>3.82%</td>
<td>0.00191</td>
</tr>
<tr>
<td>Max</td>
<td>84</td>
<td>1695</td>
<td>0.00730</td>
<td>2.13</td>
<td>100%</td>
<td>0.00569</td>
</tr>
<tr>
<td>Min</td>
<td>33</td>
<td>647</td>
<td>0.00350</td>
<td>1.02</td>
<td>90%</td>
<td>0.00009</td>
</tr>
</tbody>
</table>

By comparing the time in Table 3.6 and Table 3.7 with a 95% confidence interval, we have found that running time decreases after applying centroid reordering method.

The results are: \( \text{Davg} = 16.20; \ S^2 = 449.96; \ alpha = 0.05; \ t_{0.05, 2.9} = 2.26; \ hw = 16.00; \ lo = 0.20; \ up = 32.20. \)

Since both \( \text{lo} \) and \( \text{up} \) have the same sign, we can say that Approach 2 is faster than Approach 1 with 95% confidence. Approach 2 is faster than approach 1 by 1.3 times in the average and at the same time approach 2 is more accurate than approach 1, the average percentage of correctly assigned points increased by 2% from 96.2% to 98.2%.

3.2.5.3 Approach 3: Approach 2 + The GA Applied on a Sample Data

In a set of clustering data, usually there are many points that have close values. To save time in situations like this, a sample of the data can be taken and used as a data set for
GA to find the clusters. Also, for the extremely large clustering problems, we are forced to do this.

In `ClusteringMainProgram3.java`, a subset of the data points has been taken randomly from the whole data set. GA was run to find the clusters for this sample data. After GA found the optimal clusters for the sample data set, the rest of the points are assigned to the clusters found using the sample data. A final evaluation was done for the GA clusters over the whole data set. Three sampling rates were tried in this study. They are 20%, 10% and 5%.

**Table 3.8: Clustering Results after Applying GA on 20% a Sample Data.**

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Generation</th>
<th>The Fitness of solution</th>
<th>Fitness of Solution Divided by Fitness of actual Centroids</th>
<th>% of Correctly Assigned Points</th>
<th>Mean Squared Error between Calculated and actual Centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.06</td>
<td>588</td>
<td>0.00754</td>
<td>1.09</td>
<td>92.20%</td>
<td>0.00323</td>
</tr>
<tr>
<td>2</td>
<td>11.89</td>
<td>1055</td>
<td>0.00397</td>
<td>1.41</td>
<td>100.00%</td>
<td>0.00018</td>
</tr>
<tr>
<td>3</td>
<td>12.38</td>
<td>1081</td>
<td>0.00385</td>
<td>1.21</td>
<td>100.00%</td>
<td>0.00016</td>
</tr>
<tr>
<td>4</td>
<td>13.38</td>
<td>1180</td>
<td>0.00382</td>
<td>1.08</td>
<td>100.00%</td>
<td>0.00013</td>
</tr>
<tr>
<td>5</td>
<td>15.92</td>
<td>1402</td>
<td>0.00376</td>
<td>1.12</td>
<td>100.00%</td>
<td>0.00014</td>
</tr>
<tr>
<td>6</td>
<td>14.93</td>
<td>1309</td>
<td>0.00386</td>
<td>2.05</td>
<td>100.00%</td>
<td>0.00014</td>
</tr>
<tr>
<td>7</td>
<td>12.01</td>
<td>1052</td>
<td>0.00673</td>
<td>2.23</td>
<td>90.60%</td>
<td>0.00436</td>
</tr>
<tr>
<td>8</td>
<td>9.60</td>
<td>842</td>
<td>0.00722</td>
<td>1.99</td>
<td>89.93%</td>
<td>0.00233</td>
</tr>
<tr>
<td>9</td>
<td>15.41</td>
<td>1352</td>
<td>0.00871</td>
<td>1.14</td>
<td>92.53%</td>
<td>0.00531</td>
</tr>
<tr>
<td>10</td>
<td>12.87</td>
<td>1129</td>
<td>0.00433</td>
<td>1.13</td>
<td>100.00%</td>
<td>0.00028</td>
</tr>
<tr>
<td>Average</td>
<td>12.55</td>
<td>1099</td>
<td>0.00538</td>
<td>1.45</td>
<td>96.53%</td>
<td>0.00163</td>
</tr>
<tr>
<td>STDEV</td>
<td>2.56</td>
<td>232</td>
<td>0.00184</td>
<td>0.44</td>
<td>4.31%</td>
<td>0.00192</td>
</tr>
<tr>
<td>Max</td>
<td>15.92</td>
<td>1402</td>
<td>0.00871</td>
<td>2.23</td>
<td>100.00%</td>
<td>0.00531</td>
</tr>
<tr>
<td>Min</td>
<td>7.06</td>
<td>588</td>
<td>0.00376</td>
<td>1.08</td>
<td>89.93%</td>
<td>0.00013</td>
</tr>
</tbody>
</table>
**Table 3.9:** Clustering Results after Applying GA on 10% a Sample Data.

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Generation</th>
<th>The Fitness of solution</th>
<th>Fitness of Solution Divided by Fitness of actual Centroids</th>
<th>% of Correctly Assigned Points</th>
<th>Mean Squared Error between Calculated and actual Centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.22</td>
<td>1116</td>
<td>0.00742</td>
<td>2.16</td>
<td>91.67%</td>
<td>0.00191</td>
</tr>
<tr>
<td>2</td>
<td>4.41</td>
<td>680</td>
<td>0.00901</td>
<td>2.63</td>
<td>92.07%</td>
<td>0.00369</td>
</tr>
<tr>
<td>3</td>
<td>6.61</td>
<td>910</td>
<td>0.00788</td>
<td>2.30</td>
<td>91.87%</td>
<td>0.00478</td>
</tr>
<tr>
<td>4</td>
<td>5.30</td>
<td>783</td>
<td>0.00523</td>
<td>1.52</td>
<td>100.00%</td>
<td>0.00047</td>
</tr>
<tr>
<td>5</td>
<td>11.35</td>
<td>1713</td>
<td>0.00379</td>
<td>1.10</td>
<td>100.00%</td>
<td>0.00014</td>
</tr>
<tr>
<td>6</td>
<td>8.35</td>
<td>1261</td>
<td>0.00409</td>
<td>1.19</td>
<td>100.00%</td>
<td>0.00021</td>
</tr>
<tr>
<td>7</td>
<td>9.41</td>
<td>1377</td>
<td>0.00720</td>
<td>2.10</td>
<td>92.93%</td>
<td>0.00308</td>
</tr>
<tr>
<td>8</td>
<td>10.09</td>
<td>1463</td>
<td>0.00372</td>
<td>1.08</td>
<td>100.00%</td>
<td>0.00013</td>
</tr>
<tr>
<td>9</td>
<td>8.24</td>
<td>1218</td>
<td>0.00483</td>
<td>1.41</td>
<td>100.00%</td>
<td>0.00037</td>
</tr>
<tr>
<td>10</td>
<td>6.33</td>
<td>900</td>
<td>0.00959</td>
<td>2.80</td>
<td>93.27%</td>
<td>0.00340</td>
</tr>
<tr>
<td>Average</td>
<td>7.73</td>
<td>1142</td>
<td>0.00627</td>
<td>1.83</td>
<td>96.18%</td>
<td>0.00182</td>
</tr>
<tr>
<td>STDEV</td>
<td>2.06</td>
<td>310</td>
<td>0.00209</td>
<td>0.61</td>
<td>3.85%</td>
<td>0.00169</td>
</tr>
<tr>
<td>Max</td>
<td>11.35</td>
<td>1713</td>
<td>0.01035</td>
<td>3.02</td>
<td>100.00%</td>
<td>0.00495</td>
</tr>
<tr>
<td>Min</td>
<td>4.41</td>
<td>680</td>
<td>0.00372</td>
<td>1.08</td>
<td>91.67%</td>
<td>0.00013</td>
</tr>
</tbody>
</table>

**Table 3.10:** Clustering Results after Applying GA on 5% a Sample Data.

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Generation</th>
<th>The Fitness of solution</th>
<th>Fitness of Solution Divided by Fitness of actual Centroids</th>
<th>% of Correctly Assigned Points</th>
<th>Mean Squared Error between Calculated and actual Centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.80</td>
<td>1249</td>
<td>0.00405</td>
<td>1.18</td>
<td>100.00%</td>
<td>0.00017</td>
</tr>
<tr>
<td>2</td>
<td>6.20</td>
<td>1446</td>
<td>0.00461</td>
<td>1.34</td>
<td>100.00%</td>
<td>0.00032</td>
</tr>
<tr>
<td>3</td>
<td>3.55</td>
<td>843</td>
<td>0.00865</td>
<td>2.52</td>
<td>90.20%</td>
<td>0.00300</td>
</tr>
<tr>
<td>4</td>
<td>4.94</td>
<td>1227</td>
<td>0.00742</td>
<td>2.16</td>
<td>92.40%</td>
<td>0.00440</td>
</tr>
<tr>
<td>5</td>
<td>3.91</td>
<td>961</td>
<td>0.01035</td>
<td>3.02</td>
<td>83.13%</td>
<td>0.00459</td>
</tr>
<tr>
<td>6</td>
<td>3.33</td>
<td>804</td>
<td>0.00871</td>
<td>2.54</td>
<td>93.33%</td>
<td>0.00490</td>
</tr>
<tr>
<td>7</td>
<td>4.93</td>
<td>1189</td>
<td>0.00483</td>
<td>1.41</td>
<td>100.00%</td>
<td>0.00035</td>
</tr>
<tr>
<td>8</td>
<td>4.58</td>
<td>932</td>
<td>0.00791</td>
<td>2.31</td>
<td>89.80%</td>
<td>0.00274</td>
</tr>
<tr>
<td>9</td>
<td>4.69</td>
<td>998</td>
<td>0.00814</td>
<td>2.37</td>
<td>93.33%</td>
<td>0.00495</td>
</tr>
<tr>
<td>10</td>
<td>2.58</td>
<td>592</td>
<td>0.00905</td>
<td>2.64</td>
<td>90.20%</td>
<td>0.00262</td>
</tr>
<tr>
<td>Average</td>
<td>4.45</td>
<td>1024</td>
<td>0.00737</td>
<td>2.15</td>
<td>93.24%</td>
<td>0.00280</td>
</tr>
<tr>
<td>STDEV</td>
<td>1.06</td>
<td>240</td>
<td>0.00203</td>
<td>0.59</td>
<td>5.20%</td>
<td>0.00184</td>
</tr>
<tr>
<td>Max</td>
<td>6.20</td>
<td>1446</td>
<td>0.01035</td>
<td>3.02</td>
<td>100.00%</td>
<td>0.00495</td>
</tr>
<tr>
<td>Min</td>
<td>2.58</td>
<td>592</td>
<td>0.00405</td>
<td>1.18</td>
<td>83.13%</td>
<td>0.00017</td>
</tr>
</tbody>
</table>
By comparing the time in Tables 3.7, 3.8, 3.9 and 3.10 we found that decreasing the sampling percentage decreases the average running time. Graph 3.7 shows the relationship between average running time and sampling percentage. From this graph we found that there is a linear relationship between sampling percentage and the average running time, i.e., the time complexity of the algorithm is $O(n)$.

![Graph 3.7: The Relationship between Average Running Time and Sampling Percentage](image)

On the other hand, comparing the percentage of correctly assigned points in Tables 3.7, 3.8, 3.9, and 3.10, we found that decreasing sampling percentage causes a loss of accuracy of the results. Graph 3.8 shows the relationship between the average percentage of correctly assigned points and sampling percentage. Looking at the graph, 10% is a reasonable percentage to compare with other approaches.
When comparing approach 3 with approach 2, we used the 10% sampling percentage. By comparing the time in Table 3.7 and Table 3.9 with a 95% confidence interval, we have found that running time decreases after applying GA on a sample of data. The results are: $D_{avg} = 49.77; S^2 = 189.17 ; \alpha = 0.05; t_{0.05/2,9} = 2.26; hw = 10.37 ; lo = 39.40 ; up = 60.14$.

Since both $lo$ and $up$ have the same sign, we can say that Approach 3 is faster than Approach 2 with 95% confidence. Approach 3 is faster than approach 2 by 7.44 times in the average but at the same time approach 2 is more accurate than approach 3, the average percentage of correctly assigned points decreased from 98.2% to 96.53%. This loss of accuracy is reasonable compared to the amount of time saved using approach 3.

3.2.5.4 Approach 4: Approach 3(10%) + Recalculating Centroids
This method is applied for fine tuning the new offspring after the crossover and mutation methods. The main idea of this method is to recalculate the centroids after formation of clusters. After all data points are assigned to the closest centroids in the chromosome, the centroids are moved to the actual centers of the clusters. Using this approach, we
obtained the result as shown in Table 3.11. The main class for this run is the 

*CluteringMainProgram4.java* in Appendix B

### Table 3.11: Clustering Results after Applying GA with Recalculating Centroid.

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Generation</th>
<th>The Fitness of solution</th>
<th>Fitness of Solution Divided by Fitness of actual Centroids</th>
<th>% of Correctly Assigned Points</th>
<th>Mean Squared Error between Calculated and actual Centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.91</td>
<td>53</td>
<td>0.00352</td>
<td>1.03</td>
<td>100.00%</td>
<td>0.00009</td>
</tr>
<tr>
<td>2</td>
<td>0.93</td>
<td>52</td>
<td>0.00367</td>
<td>1.07</td>
<td>100.00%</td>
<td>0.00011</td>
</tr>
<tr>
<td>3</td>
<td>0.87</td>
<td>53</td>
<td>0.00363</td>
<td>1.06</td>
<td>100.00%</td>
<td>0.00010</td>
</tr>
<tr>
<td>4</td>
<td>0.88</td>
<td>54</td>
<td>0.00366</td>
<td>1.07</td>
<td>100.00%</td>
<td>0.00011</td>
</tr>
<tr>
<td>5</td>
<td>0.86</td>
<td>53</td>
<td>0.00356</td>
<td>1.04</td>
<td>100.00%</td>
<td>0.00009</td>
</tr>
<tr>
<td>6</td>
<td>0.84</td>
<td>53</td>
<td>0.00364</td>
<td>1.06</td>
<td>100.00%</td>
<td>0.00009</td>
</tr>
<tr>
<td>7</td>
<td>0.82</td>
<td>53</td>
<td>0.00354</td>
<td>1.03</td>
<td>100.00%</td>
<td>0.00009</td>
</tr>
<tr>
<td>8</td>
<td>0.87</td>
<td>53</td>
<td>0.00354</td>
<td>1.03</td>
<td>100.00%</td>
<td>0.00009</td>
</tr>
<tr>
<td>9</td>
<td>0.83</td>
<td>53</td>
<td>0.00352</td>
<td>1.03</td>
<td>100.00%</td>
<td>0.00007</td>
</tr>
<tr>
<td>10</td>
<td>0.83</td>
<td>53</td>
<td>0.00364</td>
<td>1.06</td>
<td>100.00%</td>
<td>0.00011</td>
</tr>
<tr>
<td>Average</td>
<td>0.86</td>
<td>53</td>
<td>0.00359</td>
<td>1.05</td>
<td>100.00%</td>
<td>0.00010</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.03</td>
<td>0.4</td>
<td>0.00006</td>
<td>0.02</td>
<td>0.00%</td>
<td>0.00001</td>
</tr>
<tr>
<td>Max</td>
<td>0.93</td>
<td>54</td>
<td>0.00367</td>
<td>1.07</td>
<td>100.00%</td>
<td>0.00011</td>
</tr>
<tr>
<td>Min</td>
<td>0.82</td>
<td>52</td>
<td>0.00352</td>
<td>1.03</td>
<td>100.00%</td>
<td>0.00007</td>
</tr>
</tbody>
</table>

By comparing the time in Table 3.9 and Table 3.11 with a 95% confidence interval, we found that running time decreases after applying recalculating centroids. The results are: $D_{avg} = 6.87; S^2 = 4.80; \alpha = 0.05; t_{0.05/2,9} = 2.26; hw = 1.65; lo = 5.21; up = 8.52$.

Since both lo and up have the same sign, we can say that approach 4 is faster than approach 3 (10% sampling percentage) with 95% confidence. Approach 4 is faster than approach 3 by 8.9 times in the average and at the same time approach 4 is more accurate than approach 3: the average percentage of correctly assigned points increased from 96.18% to 100%.  

Even though additional calculations are performed in this approach, the total running time is still lower than the previous approach. Applying this approach decreases the search space for GA, so it could find the solution in fewer generations.
3.2.5.5 Comparing the Results of the Four Approaches

From the Tables 3.6, 3.7, 3.9, and 3.11, we found that the fourth approach gave the best result in terms of time performance and accuracy of the cluster assignments of the data points.

![Figure 3.9: Run Times vs. percentage of correctly assigned points](image)

3.2.6 Applying K-Means to the Clustering Test Data

We applied the K-means method described in section 3.2.2 on the test data by executing CluterMainProgram5.java (see Appendix B). K-means method was used to compare its results to the results we have obtained using the fourth approach. The results of the K-means method are shown in Table 3.12.
Table 3.12: K-means Clustering Results for Test Data

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Cycle</th>
<th>The Fitness of solution</th>
<th>Fitness of Solution Divided by Fitness of actual Centroids</th>
<th>% of Correctly Assigned Points</th>
<th>Mean Squared Error between Calculated and actual Centroids</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.017</td>
<td>12</td>
<td>0.00734</td>
<td>2.14</td>
<td>90.87%</td>
<td>0.00217</td>
</tr>
<tr>
<td>2</td>
<td>0.021</td>
<td>15</td>
<td>0.00921</td>
<td>2.69</td>
<td>81.13%</td>
<td>0.00458</td>
</tr>
<tr>
<td>3</td>
<td>0.017</td>
<td>14</td>
<td>0.00629</td>
<td>1.83</td>
<td>90.93%</td>
<td>0.00195</td>
</tr>
<tr>
<td>4</td>
<td>0.012</td>
<td>12</td>
<td>0.01225</td>
<td>3.57</td>
<td>73.40%</td>
<td>0.01195</td>
</tr>
<tr>
<td>5</td>
<td>0.014</td>
<td>13</td>
<td>0.00615</td>
<td>1.79</td>
<td>89.87%</td>
<td>0.00309</td>
</tr>
<tr>
<td>6</td>
<td>0.013</td>
<td>12</td>
<td>0.01141</td>
<td>3.33</td>
<td>83.33%</td>
<td>0.00722</td>
</tr>
<tr>
<td>7</td>
<td>0.016</td>
<td>15</td>
<td>0.00938</td>
<td>2.73</td>
<td>83.33%</td>
<td>0.00617</td>
</tr>
<tr>
<td>8</td>
<td>0.011</td>
<td>11</td>
<td>0.00667</td>
<td>1.95</td>
<td>89.60%</td>
<td>0.00295</td>
</tr>
<tr>
<td>9</td>
<td>0.009</td>
<td>9</td>
<td>0.00964</td>
<td>2.81</td>
<td>82.67%</td>
<td>0.00583</td>
</tr>
<tr>
<td>10</td>
<td>0.011</td>
<td>11</td>
<td>0.01582</td>
<td>4.61</td>
<td>68.07%</td>
<td>0.01049</td>
</tr>
<tr>
<td>Average</td>
<td>0.014</td>
<td>12</td>
<td>0.00942</td>
<td>2.75</td>
<td>83.32%</td>
<td>0.00564</td>
</tr>
<tr>
<td>STDEV</td>
<td>0.003</td>
<td>2</td>
<td>0.00292</td>
<td>0.85</td>
<td>7.30%</td>
<td>0.00327</td>
</tr>
<tr>
<td>Max</td>
<td>0.021</td>
<td>15</td>
<td>0.01582</td>
<td>4.61</td>
<td>90.93%</td>
<td>0.01195</td>
</tr>
<tr>
<td>Min</td>
<td>0.009</td>
<td>9</td>
<td>0.00615</td>
<td>1.79</td>
<td>68.07%</td>
<td>0.00195</td>
</tr>
</tbody>
</table>

3.2.7 Comparison of the results of GA and K-Means on the Clustering Test Data

By comparing the percentage of correctly assigned points in Table 3.11 and Table 3.12 with 95% confidence interval, we have found that percentage of correctly assigned points in GA using the forth approach is higher than the percentage found using K-means method. This shows that the GA using the forth approach performs better than K-means. The results are: $D_{avg} = 0.17$; $S_2 = 0.01$; $\alpha = 0.05$; $t_{0.05/2,9} = 2.26$; $hw = 0.06$; $lo = 0.11$; $up = 0.220$.

Since both $lo$ and $up$ have the same sign, we can say that GA using the forth approach performs better than K-Means with 95% confidence. The average percentage of correctly assigned points for GA with fourth approach is 100% while the average percentage for K-means is 83.22%. However, K-means is faster than GA with fourth approach. The average run time for K-means is 0.014 minutes while for GA with fourth
approach it is 0.86 minutes. If time is more important that accuracy, we can use K-means method. Otherwise GA with fourth approach will better suit our purpose.

3.2.8 Applying GA to Gene Expression Data

The genes data is downloaded from the website of Michael Eisen's lab at the Lawrence Berkeley National Lab (LBNL) and the University of California at Berkeley (UCB) (Eisen). The data contains 6221 gene expressions in 80 dimensions. These gene expressions should be grouped into ten clusters. GA with the fourth approach was run on this data. GA algorithm was run with the following parameters, which were determined by the trial and error to obtain the best results:

- Population size = 500
- Selection method = Steady State Selection with 25% selection rate
- Crossover method = Simple point crossover with 65% crossover rate
- Mutation method = Value Mutation with 2% mutation rate
- Termination Criterion = Best Value Did not Change For 50 Generations
- Sampling percentage = 10%
- Elitism Number = 1

The main class for this run is the ClusteringMainProgram6.java in Appendix B.

Table 3.13: Applying GA with the Fourth Approach to Gene Expression Data

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>Num Of Generation</th>
<th>The Fitness of solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>184</td>
<td>65</td>
<td>0.00543</td>
</tr>
<tr>
<td>2</td>
<td>183</td>
<td>65</td>
<td>0.00543</td>
</tr>
<tr>
<td>3</td>
<td>154</td>
<td>55</td>
<td>0.00541</td>
</tr>
<tr>
<td>4</td>
<td>164</td>
<td>58</td>
<td>0.00524</td>
</tr>
<tr>
<td>5</td>
<td>198</td>
<td>71</td>
<td>0.00546</td>
</tr>
<tr>
<td>6</td>
<td>158</td>
<td>56</td>
<td>0.00559</td>
</tr>
<tr>
<td>7</td>
<td>166</td>
<td>59</td>
<td>0.00538</td>
</tr>
<tr>
<td>8</td>
<td>160</td>
<td>58</td>
<td>0.00580</td>
</tr>
<tr>
<td>9</td>
<td>154</td>
<td>56</td>
<td>0.00579</td>
</tr>
<tr>
<td>10</td>
<td>201</td>
<td>69</td>
<td>0.00541</td>
</tr>
<tr>
<td>Average</td>
<td>172</td>
<td>61</td>
<td>0.00549</td>
</tr>
<tr>
<td>STDEV</td>
<td>18</td>
<td>5.8</td>
<td>0.00018</td>
</tr>
<tr>
<td>Max</td>
<td>201</td>
<td>71</td>
<td>0.00580</td>
</tr>
<tr>
<td>Min</td>
<td>154</td>
<td>55</td>
<td>0.00524</td>
</tr>
</tbody>
</table>
3.2.9 Applying K-Means to Genes Expression

We applied the K-means method described in section 3.2.2 on the gene expression data by executing `ClusteringMainProgram7.java` (see Appendix B). We used the K-means method to compare its results to the results we have obtained using the approaches explained above. The results of the K-means method are shown in Table 3.14.

**Table 3.14: K-means Clustering Results for Gene Expression**

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>The Fitness of solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.55</td>
<td>0.00592</td>
</tr>
<tr>
<td>2</td>
<td>2.45</td>
<td>0.00627</td>
</tr>
<tr>
<td>3</td>
<td>2.40</td>
<td>0.00593</td>
</tr>
<tr>
<td>4</td>
<td>2.91</td>
<td>0.00587</td>
</tr>
<tr>
<td>5</td>
<td>2.55</td>
<td>0.00639</td>
</tr>
<tr>
<td>6</td>
<td>2.16</td>
<td>0.00593</td>
</tr>
<tr>
<td>7</td>
<td>2.40</td>
<td>0.00589</td>
</tr>
<tr>
<td>8</td>
<td>2.24</td>
<td>0.00630</td>
</tr>
<tr>
<td>9</td>
<td>2.12</td>
<td>0.00621</td>
</tr>
<tr>
<td>10</td>
<td>2.10</td>
<td>0.00619</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>2.39</strong></td>
<td><strong>0.00609</strong></td>
</tr>
<tr>
<td><strong>STDEV</strong></td>
<td><strong>0.25</strong></td>
<td><strong>0.00020</strong></td>
</tr>
<tr>
<td><strong>Max</strong></td>
<td><strong>2.91</strong></td>
<td><strong>0.00639</strong></td>
</tr>
<tr>
<td><strong>Min</strong></td>
<td><strong>2.10</strong></td>
<td><strong>0.00587</strong></td>
</tr>
</tbody>
</table>

3.2.10 Comparison of the results of GA and K-Means on Gene Expression Data

By comparing the fitness values in Table 3.13 and Table 3.14 with 95% confidence interval, we have found that fitness values in GA using the forth approach is smaller than the fitness values for K-means method. This shows that the GA using the forth approach performs better than K-means. The results are: $D_{avg} = 0.0006$; $S_2 = 3.736E-08$; alpha = 0.05; $t_{0.05/2,9} = 2.26$; $hw = 1.46E-04$; $lo = 4.50E-04$; $up = 7.42E-04$.

Since both lo and up have the same sign, we can say that GA using the forth approach performs better than K-Means with 95% confidence. However, K-means is faster than GA with fourth approach. The average run time for K-mean is 2.39 minutes.
while for GA fourth approach it is 172 minutes. If time is more important that accuracy we can use K-means method. Otherwise GA with fourth approach will better suit our purpose.

3.2.11 Comparison of the results of the adaptive GA with a known solution

Cluster Analysis and Visualization software was downloaded from Eisen lab (Eisen). The code was developed at Stanford University. In this software k-means method are used and correlation as the distance function. This software gives better fitness values than the adaptive GA with approach four as shown in Table 3.15.

Table 3.15: Esien Lab Software Clustering Results for Gene Expression

<table>
<thead>
<tr>
<th>Num of Run</th>
<th>Time in Minutes</th>
<th>The Fitness values for Eisen Lab Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.76</td>
<td>0.00465</td>
</tr>
<tr>
<td>2</td>
<td>2.32</td>
<td>0.00466</td>
</tr>
<tr>
<td>3</td>
<td>2.46</td>
<td>0.00465</td>
</tr>
<tr>
<td>4</td>
<td>3.59</td>
<td>0.00465</td>
</tr>
<tr>
<td>5</td>
<td>2.64</td>
<td>0.00470</td>
</tr>
<tr>
<td>6</td>
<td>1.57</td>
<td>0.00466</td>
</tr>
<tr>
<td>7</td>
<td>2.53</td>
<td>0.00470</td>
</tr>
<tr>
<td>8</td>
<td>3.02</td>
<td>0.00468</td>
</tr>
<tr>
<td>9</td>
<td>1.57</td>
<td>0.00467</td>
</tr>
<tr>
<td>10</td>
<td>1.25</td>
<td>0.00465</td>
</tr>
</tbody>
</table>

By comparing the fitness values in Table 3.13 and Table 3.15 with 95% confidence interval, we have found that fitness values in GA using the forth approach is bigger than the fitness values for Eisen lab software. The results are: 

\[ D_{av} = 0.00083; \]
\[ S^2 = 3.11E-08; \]
\[ \alpha = 0.05; t_{0.05/2,9} = 2.26; \]
\[ h = 0.00013; l = 0.000694; u = 0.00096. \]
Since both lo and up have the same sign, we can say that the Eisen program performs better than GA with the fourth approach with 95% confidence.

From the results of the comparison we can conclude that GA is not the most appropriate method to use in the gene clustering problem. Further research may be conducted into this area to adapt GA so that it performs better and faster with clustering problem.
The GA approach has proven to be suitable for many of the hardest computational
problems in a variety of fields, so much so that it appears as the choice of the method for
many application problems. This study applies the GA for continuous variable
optimization to two complex problems, namely the queuing network problem and the
gene clustering problem.

In many queuing network design problems, there may be many decision variables,
such as service rates and buffer sizes at the queuing stations and routing decisions. These
variables have to be determined optimally in terms of system performance, e.g., the
system throughput. There may also be some constraints on these variables, e.g., a cost
function. In order to provide a fitness value for a solution, we usually need to run a
simulation program for the network implicated by the solution because of the complex
interactions among the system components and the randomness in the services as well as
in the routing of the jobs. The simulation program can handle all the complexity in the system behavior quite easily. However, the simulation model output is random. Therefore, the simulation model needs to be run for sufficiently long time several times in order to obtain an accurate statistical measure of the fitness of a solution. Hence each run of the simulation model even with the same decisions gives somewhat different result.

The first application is a multiprogramming computer system with a CPU and three disk devices. In this continuous decision problem, we are to determine the speeds of three processors in order to maximize the throughput in the closed queuing network that allows only a fixed number of jobs. In addition, the decisions entail a difficult cost function, which is to be kept below a budget value. See section 3.1.1 for more details.

Bolch, Flesischmann and Scherppel [5] (BFS) introduced an analytical method for this problem that yields approximate solution under the assumption of the exponentially distributed service times for a small network. The BFS method was demonstrated using a queuing network of four processors and up to five jobs.

In this study, we applied the GA to deal with the continuous decisions in this problem. Every solution in the GA population is evaluated for fitness by a simulation model. We developed a simulation model for this purpose and implemented it in Java. The running time of the simulation model was a critical issue since we could only afford a small run time for each of many solutions to be generated by the GA. Fortunately, an accurate estimation of the fitness value of solution is not needed for a quick screening and comparison purposes.

In this study, we found out that the GA gives results close to the optimal solution found by the BFS method. But it does not require the complex mathematical analysis and the unnecessary assumptions about specific configuration or service time distributions. The GA proves to be a good heuristic approach to the queuing network design problem. For future research, the GA should be tested with the optimization of different queuing
network with larger number of stations and with different kinds of service time distributions.

The second problem to which we applied the GA is a gene clustering problem. In a cell, thousands of genes play regulation and control of cell activities in conjunction. After identification of the genes on the DNA, the biologist must understand the roles that these genes play in a cell in order to develop a therapeutic intervention in these activities. The biologist can measure the activity levels of each gene (gene expression) in a cell at a number discrete time points in complicated experiments under the controlled external conditions. The gene expression data produced by the thousands of genes at hundreds of time points produces an unmanageable amount of data. After analyzing this data, they have to determine a small set of genes that are actually involved in the chemical pathways in which they are interested. The generally accepted method is to use a clustering model that groups the genes that show similar expression profiles. There are many different clustering algorithms. Here we use the GA approach for continuous variables to determine the centroids of a fixed number of groups.

For this problem, a standard GA application does not yield an especially good performance. We tried four different approaches in the context of the GA. The first one was standard GA with no added features. The second one was to reorder the centroids in the chromosomes before crossover so that the centroids in different chromosomes that are close to each other appear in the same spot in their chromosomes. This method gave an improved solution over the straightforward application of the GA. In the third approach, we applied the GA on a small random sample of the gene expression data. This approach improved the GA run-time without adversely affecting the quality of the solution. In the last approach, we shifted the centroid solutions to the cluster means after the genes were assigned to the clusters. This approached further improved the performance.

In the final analysis, in order to evaluate the last GA approach, we compared the results by the K-means method, which is usually used as a standard approach to the large
clustering data. We estimated that the last GA approach produced better results than the K-means algorithm with a 95% confidence.

However, we are not completely convinced that the GA approach yields optimal results for large gene clustering problems. This area is still open for future research to improve the results. The improvement may come by reducing the number of dimensions and the number points used in determining the centroids without changing the gene expression similarities.
Appendix A

Java Documentation

The genetic algorithm is implemented in Java and consists of eight main classes. They are *Rand, Chromosome, GA, Selection, Crossover, Mutation, Evaluation* and *Termination-Criterion*. Below is the list of java documents the eight main classes and their subclasses.

**Chromosome**

```
java.lang.Object
   +--Genetic_Algorithm.Chromosome
```

The *Chromosome* class is the main element of the population. Each *Chromosome* is a candidate solution of the problem. The *Chromosome* class has a private data member of type String, called *genes*. This string consists of series of digits 0…9. This *genes* string is a concatenation of individual genes. The *Chromosome* class does not know the boundaries of a particular gene. One can set or get a particular substring of genes by using the methods of the *Chromosome* class.

**Field Detail**

```
protected double fitness
```

The fitness value for the candidate solution of this Chromosome Evaluation class is responsible for the calculation of fitness value.
public static Rand rand
    Rand class to create random numbers

protected java.lang.String genes
    This genes string is a concatenation of individual genes. The Chromosome does
    not know the boundaries of a particular gene. One can set or get a particular
    substring of genes by using the methods of the Chromosome class. This string
    consists of series of digits 0…9.

Constructor Detail

public Chromosome(int length)
    Create Chromosome class with the given length. Chromosome will create a genes
    string having random digits 0…9.
    Parameters: length - the length of the genes String

public Chromosome(java.lang.String genes)
    Create Chromosome class with given string. genes string is initialized by passing
    a string of digits.
    Parameters: genes - a string of digits

Method Detail

public void setFitness(double fitness)
    Set the fitness value.
    Parameters: fitness - the fitness value

    public double getFitness()
    Get the fitness value.
    Returns: the fitness value

    public java.lang.String getGenes()
    Get the whole genes String.
    Returns: genes

    public void setGenes(java.lang.String genes)
    Set the genes String
    Parameters: genes - the new genes

    public double getGene(int begin, int end)
    Get a gene from genes string. The range of gene value is [0,1) e.g. for
    genes="3352333", begin=2, end=4 : will return 0.52
    Parameters: begin - the boundry of genes begining inside genes string
                end - the boundry of genes end inside genes string
    Returns: a gene

    public double getGene(int begin, int end, double minScale,
                        double maxScale)
    Get a gene from genes string. The range of gene value is [0,1) e.g. for
    genes="3352333", begin=2, end=4, minScale=1, maxScale=3 : will return
    1+0.52*(3-1)= 2.04
Parameters: begin - the boundary of genes beginning inside genes string
    end - the boundary of genes end inside genes string
    minScale - min scale value for a gene
    maxScale - max scale value for a gene

Returns: a gene

public void setGene(double value, int begin, int end)
    Set a gene in genes string. The range of gene value is [0,1) e.g. for
    genes="333333", value= .522222, begin=2, end=4; genes will be "335233"
Parameters: begin - the boundary of genes beginning inside genes string
    end - the boundary of genes end inside genes string

public void setRandomDigit(int index)
    Changing a Digit Randomly. By passing the index of a digit inside the string, the
    chromosome object changes the value of this figure randomly to any value
    between zero and nine.

GA

java.lang.Object
    +--Genetic_Algorithm.GA

GA is the class that runs Genetic Algorithm. Most of GA data members are super classes.
Depending on the applications one may select the appropriate subclasses classes, e.g. one
of the data members is of type Crossover. It could be assigned any subclass such as
Single-Point-Crossover, or Two-Point-Crossover.

The algorithm starts with a set of randomly generated candidate solutions
(represented by Chromosomes) called initial generation.

For each generation, GA first copies the top best Chromosomes over to the new
population. The number of the top Chromosomes is determined by elitism-Num. The rest
of the population is constructed using the selection, crossover and mutation.

The qualified solutions (parents) are taken from the population and used to form
the next generation. This is motivated with the hope that the new population will have a
better set of solutions than the old one. The qualified solutions chosen to form a new
generation (offspring) are selected according to their fitness values. We select two
chromosomes as parents for creating new two chromosomes. Selection class is used to
select the parents. There are various subclasses of Selection class that are responsible for
different ways of selection depending on the type of the problem, e.g. Steady-State-
selection, Roulette-Wheel-Selection, etc.

After two parents are selected, we use crossover to create new two chromosomes.
There are various subclasses of Crossover class that are responsible for different ways of
performing crossover depending on the type of the problem. For a particular problem, the
user can set the probabilities of different kinds of crossovers to be applied on the
chromosomes. This is accomplished by using an array of Crossover objects. Each
element in the array is a subclass of the Crossover class and they are responsible for
different kinds of crossovers, e.g. Single-Point-Crossover, Two-Point-Crossover, etc.
These subclasses in the array are assigned a probability, which defines the frequency of applying that particular type of Crossover.

After creating new Chromosomes, mutation is applied. There are various subclasses of Mutation class that are responsible for different ways of performing mutation depending on the type of the problem. For a particular problem, the user can set the probabilities of different kinds of mutations to be applied on the Chromosomes. This is accomplished by using an array of Mutation objects. Each element in the array is a subclass of the Mutation class and they are responsible for different kinds of mutations, e.g. Value-Mutation, Add-Value-Mutation, etc. These subclasses in the array are assigned a probability, which defines the frequency of applying that particular type of mutation.

We repeat creating new Chromosomes until we reach the population size of the generation.

For each generation we evaluate all chromosomes using the Evaluation class. There are various subclasses of Evaluation class that are responsible for different ways of evaluation depending on the type of the problem, e.g. Queuing-Network, Clustering, etc. After the evaluation the chromosomes in the population are sorted according to their fitness values.

This procedure is repeated until a specified convergence condition is reached. This condition is checked by Termination-Criterion class. There are various subclasses of Termination-Criterion class that are responsible for different ways of termination criterion depending on the type of the problem. For example, a termination may occur after generating a predefined number of generations or after some steady state is reached, i.e. there is no more improvement in the best solution for a specific number of generations.

### Field Detail

**protected Evaluation evaluation**

It is responsible for evaluating the fitness values for chromosomes. For each generation we evaluate all chromosomes using the method provided by Evaluation class. There are various subclasses of Evaluation class that are responsible for different ways of evaluation depending on the type of the problem, e.g. Queuing Network, Clustering, etc.

**protected Selection selection**

Selection class is used to select the parents. There are various subclasses of Selection class that are responsible for different ways of selection depending on the type of the problem, e.g. Steady State selection, Roulette Wheel Selection, etc.

**protected Crossover[] crossover**

Array of Crossover classes. There are various subclasses of Crossover class that are responsible for different ways of performing crossover depending on the type of the problem. For a particular problem, the user can set the probabilities of different kinds of crossovers to be applied on the chromosomes. This is accomplished by using an array of Crossover objects. Each element in the array is a subclass of the Crossover class and they are responsible for different kinds of crossovers, e.g. single point crossover, two point crossover, etc. These subclasses
in the array are assigned a probability, which defines the frequency of applying that particular type of crossover.

protected `Mutation[] mutation`  
Array of Mutation classes. There are various subclasses of Mutation class that are responsible for different ways of performing mutation depending on the type of the problem. For a particular problem, the user can set the probabilities of different kinds of mutations to be applied on the chromosomes. This is accomplished by using an array of Mutation objects. Each element in the array is a subclass of the Mutation class and they are responsible for different kinds of mutations, e.g. Value Mutation, Add Value Mutation, etc. These subclasses in the array are assigned a probability, which defines the frequency of applying that particular type of mutation.

protected `TerminationCriterion terminationCriterion`  
It is used to determine when the algorithm will terminate. There are various subclasses of Termination-Criterion class that are responsible for different ways of termination criterion depending on the type of the problem. For example, a termination may occur after generating a predefined number of generations or after some steady state is reached, i.e. there is no more improvement in the best solution for a specific number of generations.

protected final int `populationSize`  
Number of Chromosomes in a generation.

protected final int `genesNum`  
Number of genes in a chromosome

protected final int `geneLength`  
Number of digits in a gene

protected final int `elitismNum`  
Elitism Number. Elitism is the name of the method that first copies the best chromosome (or a few of the best chromosomes) to the new population without crossover and mutation. The rest of the population is constructed by using GA operations (selection, crossover and mutation). Elitism can rapidly increase the performance of GA, because it prevents the loss of the best chromosomes from the previous generation. If the elitism number is large, then it could slow the process, since very few candidate solutions will be introduced in each population.

protected `Chromosome[] generation`  
Generation is an array of chromosomes, which represent solutions of the problem. We present the generation by an array of chromosomes.

protected `Rand randn`  
Rand class to create random numbers
Constructor Detail

public GA(Evaluation evaluation, Selection selection,
    Crossover[] crossover, Mutation[] mutation,
    TerminationCriterion terminationCriterion,
    int populationSize, int genesNum, int geneLength,
    int elitismNum, Rand randn)

    Creates a new instance of GA using the supplied parameters.

    Parameters:
    evaluation - Evaluation class
    selection - Selection class
    crossover - Crossover classes array
    mutation - Mutation classes array
    terminationCriterion - TerminationCriterion class
    populationSize - Number of genes in a chromosome
    genesNum - Number of digits in a gene
    geneLength - Number of digits in a gene
    elitismNum - Elitism Number
    randn - Rand class to create random numbers

Method Detail

public void run()

    Starts genetic algorithm. Begins with create First Generation then a loop of oneCycle(). This procedure is repeated until a specified convergence condition is reached. This condition is checked by Termination-Criterion class.

protected void oneCycle()

    This the core of GA which responsible for creating the next generation. First copies the best Elitism number of chromosomes over to the new population. The rest of the population is constructed using the selection, crossover and mutation. The qualified solutions (parents) are taken from the population and used to form the next generation. This is motivated with the hope that the new population will have a better set of solutions than the old one. The qualified solutions chosen to form a new generation (offspring) are selected according to their fitness values. We select two chromosomes as parents for creating new two chromosomes. Selection class is used to select the parents. There are various subclasses of Selection class that are responsible for different ways of selection depending on the type of the problem, e.g. Steady State selection, Roulette Wheel Selection, etc. After two parents are selected, we use crossover to create new two chromosomes. There are various subclasses of Crossover class that are responsible for different ways of performing crossover depending on the type of the problem. For a particular problem, the user can set the probabilities of different kinds of crossovers to be applied on the chromosomes. This is accomplished by using an array of Crossover objects. Each element in the array is a subclass of the Crossover class and they are responsible for different kinds of crossovers, e.g. single point crossover, two point crossover, etc. These subclasses in the array are assigned a probability, which defines the frequency of applying that particular type of crossover. After creating new chromosome mutation is applied. There are various subclasses of Mutation class that are responsible for different ways of
performing mutation depending on the type of the problem. For a particular problem, the user can set the probabilities of different kinds of mutations to be applied on the chromosomes. This is accomplished by using an array of Mutation objects. Each element in the array is a subclass of the Mutation class and they are responsible for different kinds of mutations, e.g. Value Mutation, Add Value Mutation, etc. These subclasses in the array are assigned a probability, which defines the frequency of applying that particular type of mutation.

protected void evaluateGeneration()
Evaluate Generation and writes the results to the chromosomes.

protected void sortGeneration()
Sorts the chromosomes in the generation according to their fitness values.

protected void addElitism(Chromosome[] nextGeneration)
Move the elite chromosomes to the next generation.
Parameters: nextGeneration - an array of Chromosomes represented next generation

protected void createOffsprings(Chromosome[] nextGeneration)
Creates offsprings for the next generation
Parameters: nextGeneration - an array of Chromosomes represented next generation

protected void Mutate(Chromosome[] nextGeneration)
Mutates the next generation after the crossover phase.
Parameters: nextGeneration - an array of Chromosomes represented next generation

def public Chromosome[] getGeneration()
Returns the whole generation
Returns: an array of Chromosomes represented the current generation

public Chromosome getBestChromosome()
Returns the best Chromosome in the current generation
Returns: the best Chromosomes

Selection
java.lang.Object
   +--Genetic_Algorithm.Selection

Selection is an abstract super class to select parents from population. There are three subclasses for selection. They are Steady-State-Selection, Roulette-Wheel-Selection, and Rank-Selection.
Field Detail

protected Chromosome[] parents
    An array represented parents for create offsprings.

protected Rand rand
    Rand class to create random numbers

Constructor Detail

public Selection(Rand rand)
    Creates a new instance of Selection.

Method Detail

public void setParents(Chromosome[] parents)
    set the parents.
    Parameters: parents - array of chromosomes that make up the parent generation

public abstract Chromosome selectParent()
    Returns a parent. The selection mechanism is implemented by subclasses
    Returns: a parent chromosome.

RankSelection

java.lang.Object
   +--Genetic_Algorithm.Selection
      +--Genetic_Algorithm.RankSelection

Rank selection class. Rank selection sorts the population according to their fitness values. Then every chromosome is assigned a rank determined by its position. The worst will have the rank 1, the second worst 2 etc. and the best will have rank N (number of chromosomes in population). The probability of selecting any chromosome is equal to its rank divided by the total of all ranks (1+2+3+...+N).

Constructor Detail

public RankSelection(Rand rand)
    Creates a new instance of RankSelection
    Parameters: rand - Rand class to create random numbers

Method Detail

public void setParents(Chromosome[] parents)
    set the parents
    Overrides: setParents in class Selection
    Parameters: parents - An array represented parents for create offspring.

public Chromosome selectParent()
    Returns a parent from the generation. Selection of the parent depends on its probability.
Specified by:
selectParent in class Selection

Returns: offspring chromosome class

RouletteWheelSelection

java.lang.Object
   +--Genetic_Algorithm.Selection
      +--Genetic_Algorithm.RouletteWheelSelection

Parents are selected with probabilities that are proportional to their fitness values. The better the chromosomes are, the more chances they have for being selected.

Constructor Detail

public RouletteWheelSelection(Rand rand,
   double qualifiedChromosomesPercentage)

Creates a new instance of RouletteWheelSelection.

Parameters:
   rand - Rand class to create random numbers
   qualifiedChromosomesPercentage - The percentage of chromosomes from the whole the population that qualify as parents.

Method Detail

public void setParents(Chromosome[] parents)

set the parents

Overrides:
   setParents in class Selection

Parameters:
   parents - An array represented parents for create offspring.

public Chromosome selectParent()

Returns a parent. Parents are selected with probabilities that are proportional to their fitness values. The better the chromosomes are, the more chances they have for being selected.

Specified by:
selectParent in class Selection

Returns: a parent chromosome.

SteadyStateSelection

java.lang.Object
   +--Genetic_Algorithm.Selection
      +--Genetic_Algorithm.SteadyStateSelection
Steady State Selection Class. In Steady State Selection, the population is sorted according to their fitness values. Good chromosomes (e.g. best 50%) are selected as parents, with the same probability for creating new offspring, while other chromosomes with lower fitness values do not qualify to be parents.

Field Detail
protected float qualifiedChromosomesPercentage
The percentage of chromosomes from the whole the population that qualify as parents.

Constructor Detail
public SteadyStateSelection(Rand rand,
float qualifiedChromosomesPercentage)
Creates a new instance of SteadyStateSelection

Parameters:
rand - Rand class to create random numbers
qualifiedChromosomesPercentage - The percentage of chromosomes from the whole the population that qualify as parents.

Method Detail
public Chromosome selectParent()
Returns a parent. Parents are selected with equal probability from the qualified chromosomes.

Specified by:
selectParent in class Selection

Returns: a parent chromosome.

Crossover
java.lang.Object
+--Genetic_Algorithm.Crossover

The Crossover class is an abstract super class that has five subclasses. They are Single-point-crossover, Two-point-crossover, Uniform-crossover, Micro-crossover and Idle-Crossover.

Single-point-crossover, Two-point-crossover and Uniform-crossover cut at the boundary of genes inside the chromosome. While Micro-crossover cuts the chromosome in any point. The Idle-Crossover returns a copy of the two input parents without any changes.

Field Detail
protected Rand randn
Rand class to create random numbers

Constructor Detail
public Crossover(Rand randn,
double crossoverProbability)
Creates a new instance of Crossover

63
**Parameters:**
- `randn` - Rand class to create random numbers
- `crossoverProbability` - probability of applying crossover on a chromosome.

**Method Detail**

```java
public double getcrossoverProbability()
```

Returns the Probability of applying crossover on a chromosome.

**Returns:**
the crossover Probability.

```java
protected abstract void crossoverOffspring(Chromosome[] parents, Chromosome[] offsprings)
```

Crossovers two parents to create two offsprings. The crossover mechanism is implemented by subclasses

**Parameters:**
parents - array of two parents.
offsprings - array of two offsprings. The function modifies this array.

---

**IdleCrossover**

java.lang.Object

+-Genetic_Algorithm.Crossover

+-Genetic_Algorithm.IdleCrossover

The Idle Crossover performs no crossover. The offsprings are exact copies of the two parents.

**Constructor Detail**

```java
public IdleCrossover(Rand randn, double crossoverRate)
```

Creates a new instance of IdleCrossover

**Parameters:**
- `randn` - Rand class to create random numbers
- `crossoverRate` - crossover rate in range [0,1]

**Method Detail**

```java
protected void crossoverOffspring(Chromosome[] parents, Chromosome[] offsprings)
```

Copies the genes of the parents to offsprings without any crossover. Therefore the offsprings will have the exact same genes as their parents.

**Specified by:**
`crossoverOffspring in class Crossover`

**Parameters:**
parents - array of two parents
offsprings - array of two offsprings. The function modifies this array.

---

**MicroCrossover**

java.lang.Object

+-Genetic_Algorithm.Crossover

+-Genetic_Algorithm.MicroCrossover
Micro Crossover class. A crossover point is selected, then digit string from the beginning of the chromosome till the crossover point is copied from the first parent, and the rest is copied from the other parent. The crossover point can be located anywhere in the string and may even split a gene.

**Constructor Detail**

```java
public MicroCrossover(Rand randn, double crossoverRate)

Creates a new instance of MicroCrossover

**Parameters:**
- `randn`: Rand class to create random numbers
- `crossoverRate`: crossover rate in range [0,1]
```

**Method Detail**

```java
protected void crossoverOffspring(Chromosome[] parents,
                                   Chromosome[] offsprings)

Creates two offsprings from two parents using micro crossover.

**Specified by:**
- `crossoverOffspring` in class `Crossover`

**Parameters:**
- `parents`: array of two parents.
- `offsprings`: array of two offsprings. The function modifies this array.
```

Single Point Crossover class. A crossover point is selected, digit string from the beginning of the chromosome to the crossover point is copied from the first parent, then the rest is copied from the other parent. The crossover point is located at the beginning of some gene in the chromosome.

**Field Detail**

```java
protected final int geneLength

Gene length
```

**Constructor Detail**

```java
public SinglePointCrossover(Rand randn, double crossoverRate,
                             int geneLength)

Creates a new instance of SinglePointCrossover

**Parameters:**
- `randn`: Rand class to create random numbers
- `crossoverRate`: crossover rate in range [0,1]
- `geneLength`: gene length
```

**Method Detail**

```java
protected void crossoverOffspring(Chromosome[] parents,
                                   Chromosome[] offsprings)

Performs single point crossover on the two parents to create two offsprings.
```
TwoPointsCrossover

java.lang.Object
+-Genetic_Algorithm.Crossover
  +-Genetic_Algorithm.TwoPointsCrossover

Two Points Crossover class. Two crossover points are selected, digit string from the beginning of the chromosome to the first crossover point is copied from the first parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent again. The two crossover points are located at the beginning of some genes.

Field Detail

protected final int geneLength
Gene length

Constructor Detail

public TwoPointsCrossover(Rand randn, double crossoverRate, int geneLength)

Creates a new instance of TwoPointsCrossover

Parameters:
  randn - Rand class to create random numbers
  crossoverRate - crossover rate in range [0,1]
  geneLength - gene length

Method Detail

protected void crossoverOffspring(Chromosome[] parents, Chromosome[] offsprings)

Performs two points crossover on two parents to create two offsprings.

Specified by:
crossoverOffspring in class Crossover

Parameters:
  parents - array of two parents
  offsprings - array. The function modifies this array.
Genes are randomly copied from the first or the second parent, to generate two offsprings.

Field Detail
protected final int geneLength
   Gene length

Constructor Detail
public UniformCrossover(Rand randn, double crossoverRate,
   Int geneLength)
   Creates a new instance of UniformCrossover
Parameters:
   randn - Rand class to create random numbers
   crossoverRate - crossover rate in range [0,1]
   geneLength - gene length

Method Detail
protected void crossoverOffspring(Chromosome[] parents,
   Chromosome[] offsprings)
   Performs uniform crossover on two parents to create two offsprings.
   Specified by:
   crossoverOffspring in class Crossover
Parameters:
   parents - array of two parents
   offsprings - array. The function modifies this array.

Mutation
java.lang.Object
   +--Genetic_Algorithm.Mutation

Mutation is an abstract super class to mutate a chromosome. There are four subclasses for Mutation. They are Add-Value-Mutation, Value-Mutation, Micro-Mutation, and Idle-Mutation.
Add-Value-Mutation and Value-Mutation change the whole value of a gene inside the chromosome. While Micro-Mutation changes a digit in the chromosome in any point. The Idle-Mutation returns a chromosome without any changes.

Field Detail
protected Rand randn
   Rand class to create random numbers

Constructor Detail
public Mutation(Rand randn, double mutationProbability)
   Creates a new instance of Mutation
Parameters:
   randn - Rand class to create random numbers
   mutationProbability - probability of applying mutation on a chromosome.
Method Detail

```java
public double getMutationProbability()
    Returns the Probability of applying mutation on a chromosome.
    Returns: the mutation Probability
```

```java
public abstract void mutate(Chromosome chromosome)
    Mutates a chromosome. The mutation mechanism is implemented by subclasses.
    Parameters: chromosome - that needs to be mutated
```

### IdleMutation

```java
java.lang.Object
    +--Genetic_Algorithm.Mutation
        +--Genetic_Algorithm.IdleMutation
```

The idle mutation performs no mutation.

#### Constructor Detail

```java
public IdleMutation(Rand randn, double mutationRate)
    Creates a new instance of IdleMutation The idle mutation performs no mutation.
    Parameters: randn - Rand class to create random numbers
                mutationRate - mutation rate in range [0,1]
```

#### Method Detail

```java
public void mutate(Chromosome chromosome)
    Performs idle mutation.
    Specified by: mutate in class Mutation
    Parameters: chromosome - that needs to be mutated
```

### AddValueMutation

```java
java.lang.Object
    +--Genetic_Algorithm.Mutation
        +--Genetic_Algorithm.AddValueMutation
```

Add value mutation adds a small random number to the value of a gene in the offspring chromosome. This small number is generated using a normal distribution with a mean of zero and a standard deviation passed by the user.

#### Field Detail

```java
protected final int geneLength
    Gene length
```
Constructor Detail

public AddValueMutation(Rand randn, double mutationRate, 
                          int geneLength, double std_dev)

  Creates a new instance of AddValueMutation

Parameters:
  randn - Rand class to create random numbers
  mutationRate - mutation rate in range [0,1]
  geneLength - gene length
  std_dev - standard deviation of the normal distribution

Method Detail

public void mutate(Chromosome chromosome)

  Performs add value mutation on an offspring chromosome. A gene is randomly
  selected from the chromosome and a value from the normal distribution is added
  to the gene. If the resulting value is less than 0 or greater than 1, then the process
  is repeated until the result is in range [0,1)

  Specified by:
  mutate in class Mutation

Parameters:
  chromosome - that needs to be mutated

MicroMutation

java.lang.Object
  +--Genetic_Algorithm.Mutation
      +--Genetic_Algorithm.MicroMutation

Micro mutation randomly changes the value of a digit (a part of a gene) in the offspring chromosome.

Constructor Detail

public MicroMutation(Rand randn, double mutationRate)

  Creates a new instance of MicroMutation

Parameters:
  randn - Rand class to create random numbers
  mutationRate - mutation rate in range [0,1]

Method Detail

public void mutate(Chromosome chromosome)

  Performs micro mutation on an offspring chromosome.

  Specified by:
  mutate in class Mutation

Parameters:
  chromosome - that needs to be mutated
ValueMutation

java.lang.Object
   +--Genetic_Algorithm.Mutation
      +--Genetic_Algorithm.ValueMutation

Value mutation changes a whole gene in the offspring chromosome to a new random value.

Field Detail
protected final int geneLength
   Gene length

Constructor Detail
public ValueMutation(Rand randn, double mutationRate, int geneLength)
   Creates a new instance of ValueMutation

Parameters:
   randn - Rand class to create random numbers
   mutationRate - mutation rate in range [0,1]
   geneLength - gene length

Method Detail
public void mutate(Chromosome chromosome)
   Performs value mutation on an offspring chromosome. The new value is generated using the uniform distribution in [0,1)

Specified by:
   mutate in class Mutation

Parameters: chromosome - that needs to be mutated

TerminationCriterion

java.lang.Object
   +--Genetic_Algorithm.TerminationCriterion

Termination-criterion is an abstract super class to check that it’s useful or not to create another new generation. This class has a subclass that is Best-Value-Didn’t-Changed-For-Many-Generation. This class Checks the best fitness value of a population that didn’t change for many generations. In my application I used the condition “fifty generation without any improvement” to stop running genetic algorithm.

Field Detail
protected long generationNo
   Counter to keep track of the generation number.

Constructor Detail
public TerminationCriterion()
   Creates a new instance of TerminationCriterion
Method Detail

public abstract boolean terminationCriterion(Chromosome[] generation)
    Returns a boolean value for termination Criterion condition. The termination
    Criterion condition is implemented by subclasses.
    **Parameters:** generation - the whole population of the chromosomes
    **Returns:** true to stop running GA otherwise false.

public abstract void reset()
    Re-instantiates the class so that it is ready to be used for a new run. The reset
    mechanism is implemented by subclasses.

public long getGenerationNo()
    Returns the generation number.
    **Returns:** generation number.

BestValueHasntChangedForManyGeneration

javafx.lang.Object
    +--Genetic_Algorithm.TerminationCriterion
        +--Genetic_Algorithm.BestValueHasntChangedForManyGeneration

Termination occurs after some steady state is reached, i.e. there is no more improvement
in the best solution for a specific number of generations.

Constructor Detail

public BestValueHasntChangedForManyGeneration(int maxNumGenerationsWithoutChangeBestSolution)
    Creates a new instance of BestValueHasntChangedForManyGeneration
    **Parameters:** maxNumGenerationsWithoutChangeBestSolution - is the max number of
        generations that allowed to continue running without improving the fitness value
        for the best solution.

Method Detail

public boolean terminationCriterion(Chromosome[] generation)
    Returns a boolean value for termination Criterion condition. The termination
    occurs if there is no more improvement in the best solution for a specific number
    of generations.
    **Specified by:**
    terminationCriterion in class TerminationCriterion
    **Parameters:** generation - the whole population of the chromosomes
    **Returns:** true to stop running GA otherwise false.

public void reset()
    Re-instantiates the class so that it is ready to be used for a new run.
    **Specified by:**
    reset in class TerminationCriterion
Rand

double lang.Object
   +--Genetic_Algorithm.Rand

The purpose of this class is to create a series of random numbers. This class uses the
formula $X_{i+1} = (a \times X_i + c) \mod m$, where ‘a’, ‘c’ and ‘m’ are constants, See 2.7. The initial
value of the seed can be different for each run, which will result in different series of
random numbers. This class has methods that can return different kind of random
numbers.

Field Detail
protected final long a
   Constant multiplier 16807l

protected final long m
   The modulus 2147483647

protected final long c
   The increment 1

Constructor Detail
public Rand()
   Instantiate a Rand class with seed=1.

public Rand(long seed)
   Instantiate a Rand class with given seed.
   Parameters: seed - initial value for seed.

Method Detail
public int randInt()
   Returns an integer random number [0,MaxInt]
   Returns: an integer random number

public long randLong()
   Returns a long random number [0,MaxLong]
   Returns: a long random number

public double rand()
   Returns random number [0,1)
   Returns: random number [0,1)

public double randNonZero()
   Returns random number (0,1)
   Returns: random number (0,1)
### Evaluation

`java.lang.Object
  +--Genetic_Algorithm.Evaluation`

Evaluation is an abstract class used to calculate a chromosome's fitness value. The method of evaluation is implemented in the subclasses.

**Constructor Detail**

```java
class Evaluation
{
  public Evaluation()
  {
    Creates a new instance of Evaluation
  }
}
```

**Method Detail**

```java
public void evaluate(Chromosome chromosome)
{
  Calculates a chromosome's fitness value and sets the chromosome's fitnessValue attribute to this value.
  Parameters: chromosome - the chromosome that needs to be evaluated
}
```

```java
protected abstract double getFitness(Chromosome chromosome)
{
  Calculates the fitness value of a chromosome. The evaluation mechanism is implemented by subclasses
  Parameters: chromosome - the chromosome that needs to be evaluated
  Returns: the fitness value of a chromosome.
}
```

### GenesClustring

`java.lang.Object
  +--Genetic_Algorithm.Evaluation
  +--Genetic_Algorithm.GenesClustring`

Evaluates a chromosome, which represent a solution for the clustering problem. The candidate solution contains a set of centroids. Evaluation calculates the fitness value, which is the sum of squared distances of the points to their corresponding clusters’ centroids.

**Constructor Detail**

```java
class GenesClustring
{
  public GenesClustring(ReadGenes readGenes, int geneLength, int clusterNum)
  {
    Creates a new instance of GenesClustring.
    Parameters: readGenes - is the genes expersions that are read from a file.
    geneLength - the length of a gene in the chromosome.
    clusterNum - the number of the clusters.
  }
}
```

**Method Detail**

```java
public double getFitness(double[][] centers)
{
  Calculates the fitness of a chromosome, which is the sum of squared distances of the points to their corresponding clusters’ centroids.
}
```
**Parameters:** centers - the centroids values from the chromosome.

**Returns:** the fitness value.

```java
public double getFitness(Chromosome chromosome)
```

Calculates the fitness of a chromosome, which is the sum of squared distances of the points to their corresponding clusters’ centroids.

**Specified by:**
getFitness in class Evaluation

**Parameters:** chromosome - the chromosome.

**Returns:** the fitness value.

```java
public void report(Chromosome bestSolution, double timeInMin, java.lang.String fileName)
```

Displays a report about the best solution. The report displays the fitness of the chromosome and the running time. If the actual centroids are given in the ReadGenes object, the report will show the fitness of the actual centroides as well, the percentage of the correctly assigned points and the mean squared error between calculated and actual centroides.

**Parameters:**
bestSolution - is the chromosome that has the best solution.
timeInMin - running time in minutes.
fileName - file name to save the result.

```java
public void reorderCentroids(Chromosome[] generation, int chromosomes_number)
```

Reorders Centroids before Cross-over Operation. see section 3.2.4.2 in thesis for details.

**Parameters:**
generation - is the array of the chromosomes in the generation.
chromosomes_number - the number of chromosomes to be ordered.

```java
public void adjustCentroids(Chromosome chromosome)
```

This method is applied for fine tuning the new offspring after the crossover and mutation methods. The main idea of this method is to shift the centroids to the means of the current clusters. After each point is assigned to the closest centroid in the chromosome, the centroid is shifted to the actual center of the points grouped around this centroid. See section 3.2.4.4 for more details

**Parameters:** chromosome - the chromosome that need to centroide adjustment

---

**QueuingNetwork**

java.lang.Object
   +--Genetic_Algorithm.Evaluation
      +--Genetic_Algorithm.QueuingNetwork

This class is used to evaluate a chromosome, which represents a candidate solution for Queuing Network problem. The decision variables are the speeds of the processor and the disks that will maximize the fitness value expressed as a
composite function of the throughput and the cost. In order to obtain a fitness value for the queuing network design problem, a simulation program is executed for this queuing network to estimate the throughput.

**Constructor Detail**

public `QueuingNetwork(double testTime, double costFactor, int geneLength, Rand randn)`

Creates a new instance of QueuingNetwork

**Parameters:**
- `testTime` - the running time for the simulation model.
- `costFactor` - the cost penalty factor. See section 3.1.4.
- `geneLength` - the length of a gene in the chromosome.
- `randn` - Rand class to create random numbers

**Method Detail**

protected double `getFitness(Chromosome chromosome)`

Calculates the fitness value of a chromosome. To calculate the fitness value of a chromosome a simulation model needs to run. The result of this run is the throughput.

**Specified by:**
- `getFitness` in class `Evaluation`

**Parameters:**
- `chromosome` - the chromosome that needs to be evaluated

**Returns:**
- the fitness value of a chromosome.

public void `report(Chromosome bestSolution)`

Displays a report about the best solution. The report displays the speeds of the processor and the disks and the throughput.

**Parameters:**
- `bestSolution` - is the chromosome that has the best solution.

public void `setTestTime(double testTime)`

Sets running time for the simulation model.

**Parameters:**
- `testTime` - the running time for the simulation model.
Appendix B

Files

The attached CD contains 58 classes implemented in java and three data files. The first data file is Genes5.in. This file contains the test data for clustering problem and used by ClusteringMainProgram1.java, ClusteringMainProgram2.java, ClusteringMainProgram3.java, ClusteringMainProgram4.java and ClusteringMainProgram5.java. The second data file is yeastall.in. This file contains genes expression data and used by ClusteringMainProgram6.java, ClusteringMainProgram7.java.

The third data file, yeastall_public.txt, was downloaded from Eisen Lab web site[13]. This file is a text file for genes expression data that is not in a readable format by our program. The converted file for the same data is yeastall.in, which is in the format that can be read by the Java programs we have developed.
References


