A More “Realistic” Genetic Algorithm

-----CGA

A thesis submitted to the Cardiff University

For the degree of

Master of Philosophy in Engineering

By

Hui Chen

February 2013

Acknowledgements

I am heartily thankful to my supervisors Professor John Miles and Dr Alan Kwan for giving me the opportunity to carry out this study and encouraging me a lot during these years.

Then, I offer my regards and blessings to all of those who helped and supported me in any respect during the completion of the project; Dr Haijiang Li, my parents Jianying Yang and Xueping Chen and all my friends.

Last but not least, I also wish to thank Dr Michael Packianather, Dr Shuqi Pan and all the staffs in the research office for their patience and support.

Hui Chen

2013

Abstract

Genetic Algorithms (GAs) are loosely based on the concept of the natural cycle of reproduction with selective pressures favouring the individuals which are best suited to their environment (i.e. fitness function). However, there are many features of natural reproduction which are not replicated in GAs, such as population members taking some time to reach puberty. This thesis describes a programme of research which set out to investigate what would be the impact on the performance of a GA of introducing additional features which more closely replicate real life processes. The motivation for the work was curiosity. The approach has been tested using various standard test functions. The results are interesting and show that when compared with a Canonical GA, introducing various features such as the need to reach puberty before reproduction can occur and risk of illness can enhance the effectiveness of GAs in terms of the overall effort needed to find a solution. As the method simulating the nature rules, Cardiff Genetic Algorithm (CGA) introduces several features to each individual in programming modelling the real world. Each individual of the population is given a life-span and an age, the population size is allowed to vary; and rather than generations, the concept of time steps is introduced with each individual living for a number of time steps. An additional feature is also discussed involving multiple populations which have to compete for a limited resource which can be thought of as "water". This together with an illness parameter and accidental death are used to study the behaviour of these populations.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Introduction

Genetic Algorithm (GA) is a computing model simulating Darwin's natural selection and biological evolution. As a search procedure of searching optimal solutions by simulating natural evolution process, which was first published in an influential monograph Adaptation in Natural and Artificial Systems by Professor J. Holland in Michigan University in 1975, in the United States. While GA has gradually become well-known and has been further developed, the algorithm which Holland originally put forward is usually called simple Genetic Algorithm (in the following text, it is called Canonical GA for distinguishing). Since the founding of Canonical GA, a large number of studies on genetic algorithm have emerged. Applications include varying areas, such as job shop scheduling, training feed forward neural network, image feature extraction, and image feature recognition (Buckles and Petry, 1992).

This study comes from the research on the history of GAs. One question unresolved is what would happen if one form of GA evolved more closely to mimic natural species. Apart from developing more applications for modern GAs, will it be of more value if we go back to look at the original idea of GA? Hence, Cardiff Genetic Algorithm (CGA) does not evolve from a desire to improve the performance of GAs but rather evolves more closely mimicking real life. In order to achieve this curious aim, this study has the following objectives: identify different frameworks of CGA apart from Canonical GA; introduce different species models used in testing and operate tests of several functions to show the performance of CGA with single population; summarize the parameter elements' effects in CGA; and extend CGA to the two-population method.

Purely by chance, it was found that the CGA does perform better than Canonical GA on certain test problems. And tests are undertaken to determine whether this enhanced performance can be seen in other problems. All the new framework and parameters used in this method is trying to give a new way of developing GA, not only from the operation itself, but also the fundamental principal of GA. Maybe it is not as perfect as what has been developed in normal GAs, but it is still worth trying.

Generally speaking, the research methods applied in this study include combining history research, statistics and data research, computer programming and theory study all together. Books, journal research results and online resources are very important to this research. The selection and functions of modelling parameter's value used in this study are based on the results of previous researches (e.g. different birth-rate and breading ages of the modelling species). Theory study is focus on all the section.

1.2 Arrangement of Thesis

There are six chapters in this thesis. Following this introductory chapter, the subsequent chapter introduces the background knowledge on Genetic Algorithms, including the basic theory and different improvement approaches during the last three decades. Chapter 3 gives details about the Cardiff Genetic Algorithm (i.e. the genetic algorithm developed in this work). New parameters and framework will be explained and behaviours of CGA with different models will also be shown in that chapter. Comparison between Canonical GA and CGA using three different functions is presented in Chapter 3 as well. In Chapter 4, some elements effects for the CGA performance will be discussed in details. A new edition of CGA, two-monkey CGA (TM-CGA) will be introduced and compared with Single CGA in Chapter 5, which is the multiple population attempt of CGA. In the final part of the thesis, the concluding chapter summarizes the work completed and offers expectations for future research.

Chapter 2 Genetic Algorithms

2.1 Chapter Introduction

This chapter will mainly introduce the theory of genetic algorithms. The following sections mainly cover the subjects below:

- Background knowledge of GA
- Methodology for GA
- Important elements and parameters in GAs

And at last, there is a review of the development of GAs these years.

2.2 Background of Genetic Algorithms

Computer programs, generally, are predictable. They start from a point 1 and after some specific path reach a point 2. (Ladd, 1996) However, as development of robotics has shown, it is not enough to just use deterministic algorithms in software, what is needed in some circumstances is the ability to learn and cope with stochastic.

"If you're looking for a paradigm of adaptability, look no further than biology" (Ladd, 1996). Throughout billions of years, living things have absolutely shown their flexibility and adaptability in the changing environment. As Darwin observed and concluded, the process of Natural Selection is the mechanism by which evolution takes place in the population of specific organisms. According to that rule, the evolutionary process of change happens in all forms of life over generations, and

Figure 2.1: Image of DNA

evolutionary biology is the study of how evolution occurs. An organism inherits features from its parents through genes. Changes (also called mutations) in these genes can produce a new trait in the offspring of an organism. If a new trait makes these offspring better suited to their environment, they will generally be more successful at surviving and reproducing (Chiras, 2006). It causes useful traits to become more common. Over many generations, a population can acquire so many new traits that it may become a new species. Thanks to the discovery of deoxyribonucleic acid (DNA) molecule in 1951 by biologists Crick and Watson (Ladd, 1996), the mysterious agents behind evolution are known to be mixing and sifting gene pools, acting on variations produced by the differential reproduction and mutation of genotypes. As Kehoe said in her book (1998), population can be thought of as gene pools, while genes are chemical compounds, segments of long strings of DNA which can be called chromosomes. An individual's set of genes is their genotype. The image of DNA is like the picture shown in Figure 2.1.

2.3 Methodology of GAs

Genetic Algorithms (GA) are designed to simulate Darwinian natural selection and biological evolution and, and hence to be utilised to search for optimal / high performance solutions. Learning from the adapting life species to uncountable niches in an ever-changing environment, software, in turn, can simulate natural techniques in helping optimization and search problems to evolve towards better solutions. In 1975, John Holland defined the concept of Genetic Algorithms in his paper (Holland, 1975).

According to Mitchell (1998), the main idea of a GA is to set a number of initial solutions that reproduce based on their individual fitness in a given environment. The evolution usually starts from a population of the randomly selected individuals. In every generation, the fitness of each individual will be calculated though certain fitness functions. Multiple individuals will be stochastically selected from the current population as parents, according to their fitness, and subjected to crossover (exchange part of the genes in parents' chromosomes) to produce offspring which becomes the next generation. Then the new population of the next generation is modified and recombined (possibly after randomly mutation). The details of each operator of a GA are discussed in the following section.

The feature which is deemed to be most critical is the concept of "the survival of the fittest" where selective pressure on the members of the population is applied to ensure that the fittest members of the population, with the fitness being determined by an appropriate function, have the greatest chance of reproduction and hence passing their genes successfully into the next generation. Typically, each individual of the population exists for a "generation" at the end of which the reproductive cycle takes place with all members of the population being considered for reproduction with selection being stochastically linked to fitness. Often "elitism" is used with a fixed percentage of the fittest members of the population being passed through to the next generation. This reproductive cycle is nothing like that of many higher order life forms which have a population which consists of members of varying ages, some of which are fertile and some are not. In such populations, the times at which breeding occurs can vary considerably. Also, such organisms are subjected to features like competition for food from other life forms and illness.

Figure 2.2 illustrates the framework of GA with explanation given as follows.

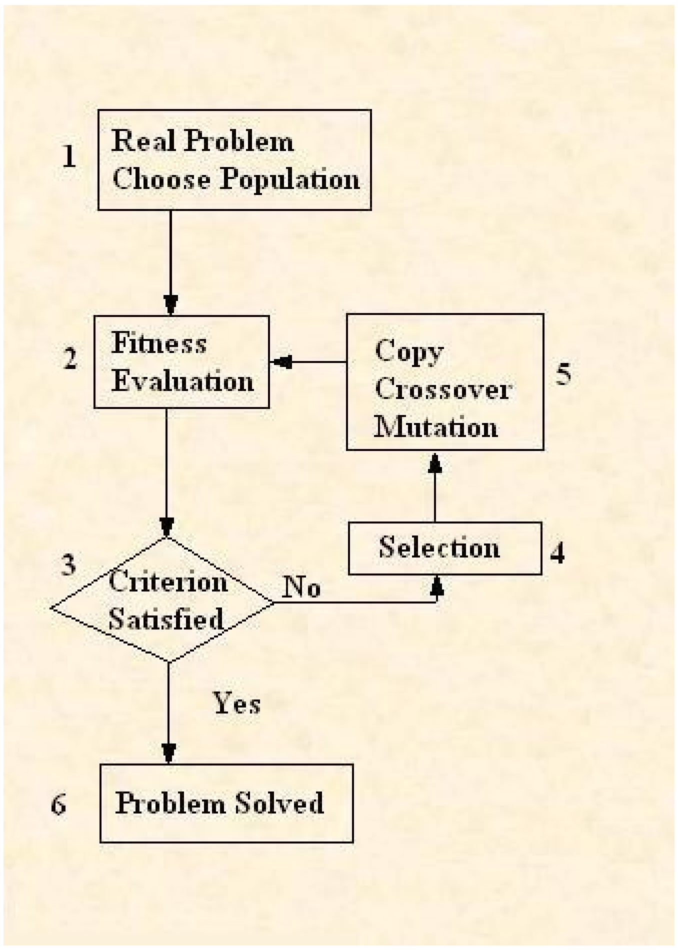


Figure 2.2: Flow diagram of simple genetic algorithm

The procedure of GA (as shown in Figure 2.2) consists of 6 essential steps:

1 Set initial population size for the real problem and create initial random population.

2 Calculate the fitness value for each member of the population based on its evaluation against the current problem.

3 Check all the fitness with criterion satisfied function.

If ‘No’,

4 Select parents by fitness and solutions with higher fitness value are most likely to be parent during reproduction.

5 Copy, crossover & mutation to produce the new offspring then replace the weak individuals by the new solution, thus one generation is complete.

Back to step 2.

If ‘Yes’,

6 Problem has been solved.

Generally speaking, when confronting a real search problem, the search parameters or variables need to be encoded, and represent the problem as a function objective or target with which we can decide termination of the run because the objective has been met. Each individual possible solution is like a chromosome in the real life species. Once we have the genetic representation and the fitness function (i.e. objective function) has been defined, GA proceeds to initialize a population of solutions randomly (step 1). After calculated the fitness for every individuals (step 2), all the fitness of the population will be checked by the criterion satisfied function (step 3). If there is any acceptable solution (If 'Yes'), the loop will be finish (step 6), otherwise (If 'No') GA will improve the population through repetitive application of parent selection (step 4) and reproduction (including copying, crossover, mutation in step 5). The population in the new generation includes two parts: the good fitness individuals in the last generation and new offspring produced by crossover and mutation. More details about the elements in this process will be discussed in next pages.

2.4 The Operators of GAs

As mentioned in section 2.3, GA consists of the following components:

1. Population of chromosomes.
2. Selection according to fitness.
3. Crossover to produce new offspring.
4. Random mutation of new offspring.

In the following pages, these elements will be described with more details.

2.4.1 Chromosome

The chromosome is typically represented by a form of binary bit string in a GA population. Each chromosome represents a point in the search space of all the candidate solutions. During the GA process, variation in the populations of chromosomes takes place constantly, by replacing one such population with another. These chromosomes are also referred to as genotypes, structures, strings or individuals. Although chromosomes is not just the problem of coding, there is also the representation (i.e. what specific feature of the problem being solved is included in the string). Representation is vital because if it is not flexible enough, the search is restricted and it may be impossible for the algorithm to find good solutions.

2.4.2 Fitness Assessment

The fitness level of each chromosome is assessed with a score which is calculated through a certain fitness function. Unlike the objective functions that are used in traditional optimization, fitness function in GAs is unable to take constraints into account, which are typically included in the objective as penalty functions. GAs' search will start with relatively weak information on performance and still reach good results. Balling (2003) has compared the performances using different kinds of fitness function. Comparison result of a simple test function influenced by sundry fitness function is presented in Chapter 3.

Since the fitness for each chromosome is defined as the ability of a chromosome to solve the problem. One of the common applications of GAs is optimization, which aims to identify a set of parameters that maximize or minimize a complex multi-parameter function.

As a non-numerical example, we can consider elements to be taken into account in the concept design phase of a modern vehicle. These elements include length, height, weight, power, safety coefficient, time of production, cost of production etc. GA solves the problem by defining a 7 parameters function. Each parameter can be encoded as a 10 bit string for example, and the chromosome is made up of 70 bit strings which comprises and combines the 7 parameter. What would such a fitness function mean? How "good" is a car? Can how "good" a car be calculated by some formula involving these 7 numbers? Or indeed, involving 700 numbers? As a common example, we cannot ask for a large, powerful car with a high safety coefficient as well as a cheap cost and to be produced quickly at the same time. Furthermore, there is a limitation for some parameters: a car cannot be as long as a train, so it has length limitation; it must be suitable for a person to sit in, so it has height limitation. The fitness for this problem is not fixed; it depends on the requirements from individual designers or customers. For instance, if the producer wants the cheapest car, then the cost of production should be the most important element to be considered in the fitness function. If the customer wants a car to be comfortable and safe, then the fitness function should contain both the size parameters (height, length) and the safety coefficient. Of course, all the parameters have linked relationships between each other: when the car is large and fast, obviously it should be more expensive than a small and slow one. Thus, how to define the fitness function sometimes can be the most difficult but critical part in a multi-objective optimisation planning.

2.4.3 Selection

Generally, there are three types of operators involved in the simplest form of genetic algorithm: selection, crossover, and mutation.

Selection operator selects parents according to their fitness (Goldberg, 1989); solutions with higher fitness values are more likely to be parent of new solutions during reproduction. All selection functions are stochastically designed so that a small proportion of less fit solutions are possible to be selected. This helps to keep the diversity of the population large, preventing premature convergence on poor solutions. Popular and well-studied selection methods include tournament selection and roulette wheel selection.

2.4.3.1 Tournament selection

There are two stages for tournament selection: the first step is to select a group of N ($N \geq 2$) individuals; the second step is to select the individual with the highest fitness from the group to carry out the crossover and mutation, simultaneously all other individuals remain in the gene pool waiting for the next loop of selection. Selection pressure can be easily adjusted by changing the tournament size. Tournament size is inversely proportional to the chance of selection, e.g., with larger tournament size, weak individual has smaller chance to be selected, contrariwise. Tournament selection has several advantages: it is efficient to encode, and allows the selection pressure to be easily adjusted.

The method used in the work of this thesis is tournament selection. However, there is a subtle difference. In normal tournament selection, the selected members are discarded from the selection candidates for the next selecting run, i.e. they are not available for selection when they have already been selected. But in this work, the rule is followed only by the parents that are paired up for crossover, to avoid cloning a child from a single parent. For selecting the next pair of parents, the individuals from the previous selection are still available for the next tournament so that all the individuals can potentially be selected several times in each generation.

2.4.3.2 Roulette wheel selection

As another kind of fitness-proportionate selection, roulette wheel selection is also used in Genetic Algorithms to select potentially useful solutions for recombination (crossover). Generally, candidate solutions with a higher fitness will be more likely to be selected comparing to the poorer fitness individual. However, there is still a chance that they might not be. On the contrary, with fitness proportionate selection, there is also a possibility for some weaker solutions that they may survive in the selection process. This is an advantage: although an individual is weak among the whole population in one generation, it may include some components which could be proved useful in the following recombination process; and through this selection rule, those possible useful individual components might be passed through to the following generation. As with the tournament selection method, the basic roulette wheel selection method is also stochastic sampling with replacement (Haupt R and Haupt S, 2004).

The analogy to a roulette wheel can be envisaged by imagining a roulette wheel in which each candidate solution represents a pocket on the wheel; the size of the pockets is proportionate to the fitness and also to the probability of selection. The probability of selecting N chromosomes from the population is equivalent to that of playing N games on the roulette wheel, as each candidate is drawn independently (as shown in figure 2.3 shown). (Zalzala and Fleming, 1997)

Individual	Fitness	Proportion
1	1.0	1/8
2	2.0	1/4
3	1.5	3/16
4	0.5	1/16
5	2.5	5/16
6	0.5	1/16

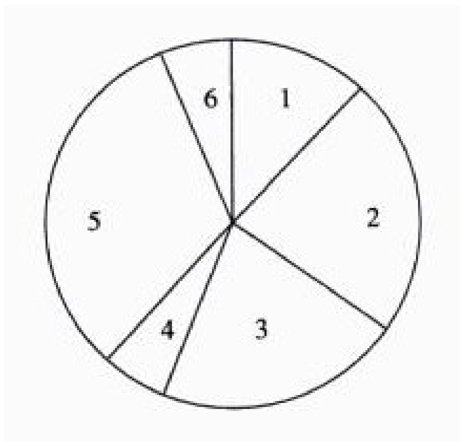


Figure 2.3: the roulette wheel selection strategy

2.4.4 Crossover

This operator is the key of Genetic Algorithms' power. There are several forms of crossover: e.g. n-point crossover and uniform crossover (Reeves, 1994; Eshelman, 1991). Taking the one-point crossover as an example, it randomly chooses a point in two bit strings and exchanges the subsequence to that locus between two parents to produce two offspring. As Holland's building blocks hypothesis (1975), the crossover operator imitates the rule of biological recombination. Without crossover, the species reproduction could have a population containing successful members' gene from one or the other parent (and also be effect by the mutation), however, no member has both. With crossover, beneficial mutations on two parents can take place immediately when they reproduce.

Nevertheless, crossover as well as being beneficial in bringing two useful components together can also be disruptive and break up a good individual, but overall, the selective pressure in GAs is positive. Crossover can be particularly disruptive when the solutions are close to the best solution, because it is almost inevitable that it will disrupt good solutions and in such cases mimetic algorithms tend to be used for the final stages (Grefenstette, 1993; De Jong 1975, and Spears, 1993). If the more successful parents are selected more often than the less successful ones and crossovers occur, the offspring are naturally adapted to the environment, like native species.

2.4.5 Mutation

This operator randomly chooses a point in one bit-string and changes the bit from 1 to 0 or from 0 to 1 with the program using binary coding. Mutation can occur at each bit position in a string with some small possibility but it is a competitively intensive way to effect mutation.

The way using in the program of this thesis, was one point mutation method. One decimal figure between 0 and 1 was random chosen to each individual. If that figure was below the value of mutation rate, a location is randomly chosen and mutation takes place. In this case, every individual has only 1 location to change from 0 to 1 or 1 to 0.

2.5 Development GAs

The classes of algorithms known as evolutionary algorithms are grouped together because they are said, in some way to mimic natural process. So for instance Particle Swarm (Kennedy and Eberhart, 1995) is said to replicate the behaviour of flock of creatures such as birds, Ant Colony algorithms (Dorigo, etc. 1996.) are intended to replicate the search mechanisms of ants and more recently there has been the eponymous Bee Algorithm (Pham, etc. 2007). Probably the most widely used and researched of all Evolutionary Algorithms is the Genetic Algorithm first introduced by Holland (1975) and since widely applied by many (e.g. Goldberg 1989, Machwe and Parmee 2007, Coelho 2002). With all of these algorithms the way in which they replicate the natural processes of the relevant organism is fairly limited with the search mechanism typically focussing on one particular feature which is deemed to be that which gives the search properties and mostly ignoring all of the other features because these are thought to be irrelevant.

In the past four decades, many researchers have examined various different methods to improve the efficiency or effectiveness or widen the applications of GAs. Some of them involved multiple population (parallel GAs) (e.g. Cant's-Paz, 1997, 1998, Fogarty and Huang, 1991), and others improved the structure by introducing features such as new parameters and encoding (Hu, 2008, Schraudolph and Belew, 1992 and Yao and Liu, 1998). For the GA operators, many studies appreciate the perspective that reproduction methods which based on the involvement of two parents are more "biology inspired”. Some researches suggest that "parents” more than two generate chromosomes of higher quality (Eiben, 1994 and Ting, 2005).Opinions are divided over the importance of crossover and mutation. There are many references in Fogel’s research (2006) supports the importance of mutation-based research. Although crossover and mutation are known as the main genetic operators, it is possible to use other operators such as regrouping, colonization-extinction, and migration in genetic algorithms (Akbari, 2010). Considering Gas are a sub-field of Evolutionary Algorithms, attention should be paid to research developments in EA, such as Swan intelligence which includes Ant colony optimization (Colorni, 1991 and Prabhakar, 2012), Particle swarm optimization (Rania, 2005), and intelligent water drops or IWD algorithms (Hamed, 2009), and other evolutionary computing algorithms including Bacteriologic algorithm (Baudry, 2005), Differential search algorithm (Civiciogly, 2012), Gaussian adaptation (Kjellström, 1991) and etc. What’s more, Multiple Objective Optimization in Evolution Algorithm has been introduced to GAs study(Keeney, 1976, Deb, 1999 and 2002, Coelho 2007). All of these researches quite logically, focussed on various aspects of the algorithm. They take a very structured approach rooted in mathematics rather than natural processes .It needs to be acknowledged that the study has achieved impressive results.

The work which is described in this paper resulted not from a desire to enhance the performance of GAs but rather from a curiosity to investigate what would happen if the behaviour of a GA was made to more closely mimic the behaviour or various life forms. GAs are meant to replicate natural behaviour so rather than just selecting those features of natural behaviour which are deemed to be desirable, the aim has been to selectively introduce other features, which have no obvious advantage and to investigate the impact of these features. So this is research driven by curiosity rather than having a definite end goal.

What to use to test any search algorithm is always a difficult choice, especially when one considers the "no free lunch theorem" (Wolpert and Macready 1997). In this work, the approach used has been to use various standard test functions. It is recognised that these have particular features and do not replicate all the features that one finds in highly complex "real life" fitness functions, but they are easily replicated by others, thus offering the chance of checking the results presented and the choice of any other fitness function would be equally arbitrary and limited. If one is to compare the impact of a change in the structure of a GA, it is necessary to have a benchmark against which the impact of the change can be measured. The obvious choice is the canonical GA (Goldberg, 1989) and this has been used throughout this work. It is recognised that the new features which are being investigated may behave very differently in other forms of GA but as there are many of these, to undertake this sort of testing is not feasible within a sensible time span.

Throughout this work, the new form of GA is referred to as the CGA (Chen, etc. 2008) This is still very much a work in progress with work having started some four years ago but the results have reached a sufficient stage of maturity for them to be subjected to necessary process of peer review and scrutiny by the wider research community. As will be shown in the following chapters of this thesis, there are some unexpected results.

2.6 Chapter Summary

In this chapter, history and development of GAs are introduced. We also got a brief idea of how ordinary GA works and the concept of important operators used in GAs. Chromosomes definition, fitness assessment, parents selection and, of course the key point of GA, crossover and mutation, are all given in great details. In later chapters, these main operators will still be used as in other GAs but there are many new things added for the new GA. Chapter 3, 4, and 5 will give more information of CGA as our GA method.

Chapter 3 Single Population CGA

3.1 Chapter Introduction

The brief idea of single population CGA will be described in this Chapter. As a method which is simulating the natural rules, CGA introduces several features to each individual during programming, modelling the real world.

A fundamental new framework of single population CGA will be introduced in the beginning of this chapter. After that, several new parameters, such as time-step, age, life-span, breeding age and effective-fitness, etc., which have been used to help the population mimicking will be listed and described with more details. Following the parameters explained, three different species models using different parameter values will be described in 3.4. In addition, there is a description of how does single population CGA behave after model comparison. Test programs were used in this chapter to compare the performance of CGA and Canonical GA with three optimization functions. Chapter summary is the last part in this chapter.

3.2 New Framework of Single Population CGA

Simulating the population in the real world is the basic thought of Genetic Algorithms. The governing framework for the evolutionary processes is based on the initial idea of John Holland from 1975: two selected parents create new offspring which keeps their good genes remained in the whole population gene pool; later individuals replace former ones and the population continue generation by generation. However, in Canonical GA, later generation replace previous generation by a high percentage (mostly over 60%) which means that the least fit individuals would be slowly eliminated. Indeed, the original concept of GA was a method from which real world biological operations were derived. In nature, every creature has a certain life span, but only some of the individual can survive and continue their gene pool. Furthermore, there is no simultaneous mass replacement of the population from one generation to the next. Rather, the generations overlap. The current work investigates whether a GA method based on a more natural governing framework is possible,

CGA is the result of the above considerations. It is a new approach which is closer to the natural evolution than what is currently available. The principal difference between the CGA framework and other GAs is the ability of survival between generations. Other features include delayed reproduction and the creation of a dynamic population where individuals within the population do not need to be all replaced in every single iteration.

New method in CGA include assigning each individual a certain “life span”, which means no matter how poor the fitness is, it would still stay in the population for a length of time. CGA also operates in “time-step” rather than “generation”. Another aspect of CGA related to the life span is that each individual has its own progressive “age”. To simulate nature, the initial ages of every individual are randomly chosen. In this way, a big difference in CGA is that only a small number of individual will be selected as parents for crossover, whereby probably just three or four children will be created per time-step in a population of 1-100. In additional, a new parameter of “effective fitness” is introduced for the parents’ selection. All the parameters mentioned above will have further explanation later in Chapter 3.3.

By accepting these brief new ideas, the framework of CGA will be easier to understand. As shown in Figure 3.1, the framework with single population can be presented like this:

The process starts with the creation of the initial population. Each individual is given an initial age, fitness, effective fitness and a life-span. The process then starts to work around the main loop (Fig. 3.1). This is shown as multiple lines because the bulk of the population works its way around the loop once in each time step.

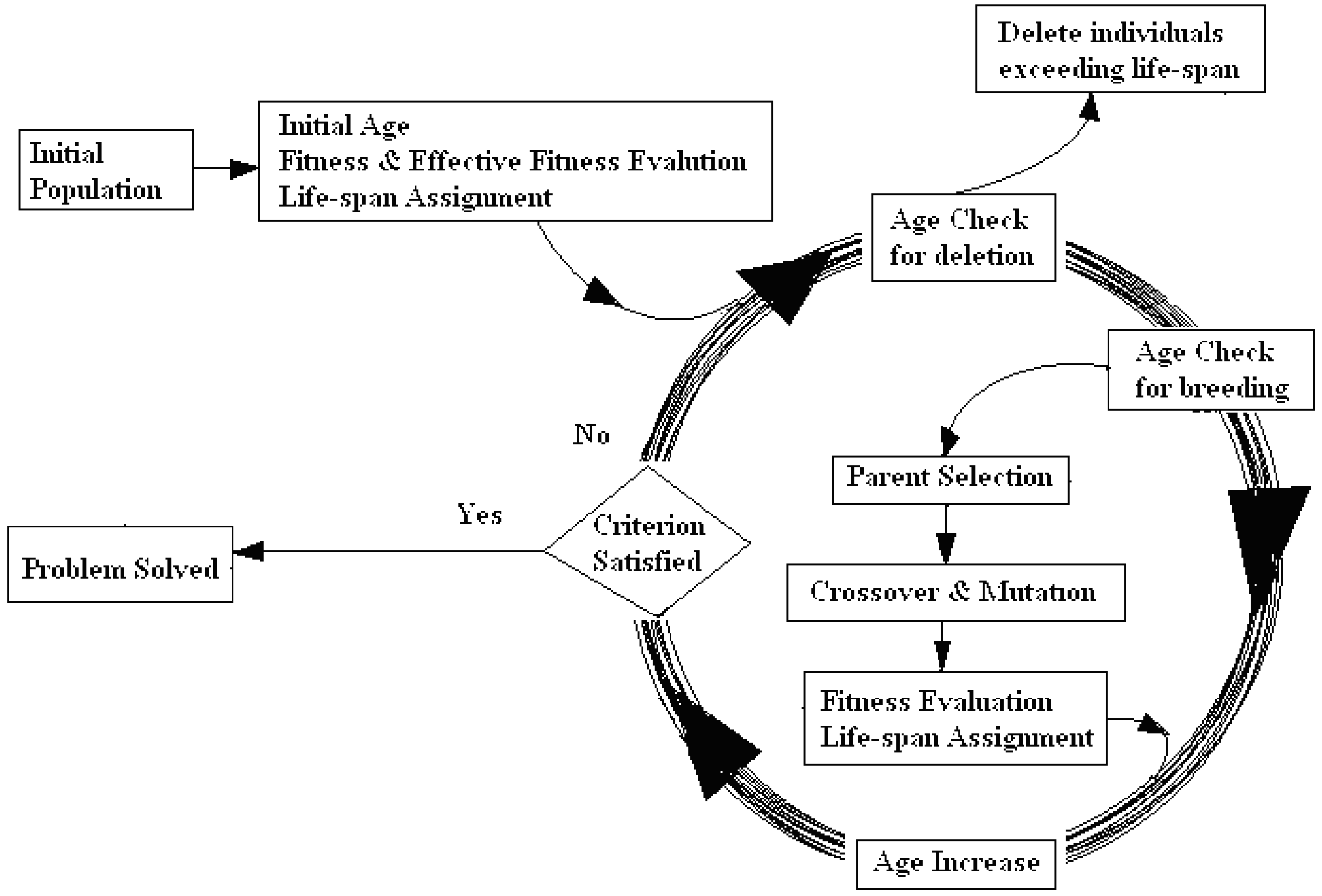


Figure 3.1: Framework of Single CGA

The whole population has its age checked first, with over-age individuals being deleted. Next, the remaining individuals have their age checked to see if they are old enough to be considered for breeding. Those who are of sufficient age are considered for the parent gene pool. The selection process uses tournament selection but only a small fraction of population is selected, the number chosen being determined by the population creation rate. The rest of the population continues around the main loop. The selection of just a fraction of the fertile individuals is closer to a real population where not everybody reproduces at the same time. The population creation rate is chosen to keep the population to the required size (Chapter 4). The breeding processes of crossover and mutation have been implemented as they would be in a canonical GA.

After evaluating the new offspring's fitness and assigning their life-spans (their initial ages will be set as 0), all individuals in the population increase their ages by an incremented of 1. There is then a check at the end of the loop to see if the finishing criterion satisfies checking. If this is not the case then the process is repeated.

To summarize, there are nine steps in the CGA:

```
1. Set initial population size, and maximum number of time-steps; create initial random population, give random age and life-span to each individual; calculate the fitness and effective fitness values for each member of the population.
2. Age checking for life-span exceeding deletion
3. Accident deletion
4. Age checking for breeding
5. Select parents according to population creation rate and population size

6. Crossover & mutation to produce the new offspring and set age as 0
7. Evaluate fitness and assign life-span for offspring
8. All population members including both old and new increase age by 1
9. Check all the fitness with Criterion Satisfied Function
If "No", back to step 2
If "Yes", problem has been solved
```

3.3 New Parameters in CGA

As stated above, the aim of the CGA is to assess the impact of the introduction of various additional aspects of natural life into a GA. There are many aspects of real world behaviour in life forms which could be introduced, so the decision arises as to what features should be investigated and also what life forms should be mimicked. There are some features which apply to all life forms, such as diseases, the risk of food shortages and typical life span. There are also significant differences between, for example, plants which typically produce huge numbers of offspring, most of which fail to reproduce or are eaten while immature, and mammals which produce far fewer offspring but have higher survival rates. In this work, the underlying thinking has concentrated more on aspects of the life of fauna rather than flora.

The question then arises as to what aspects should be investigated. The choices that have been made are necessarily, to a degree, arbitrary but concern significant aspects of the life of animals that are not included in current GAs.

The new features that have been investigated are:

3.3.1 Time-steps:

In a Canonical GA, generation is used to count the number of iterations, whereas CGA uses time-step for the same effect. Real life populations typically contain individuals of differing ages; different levels of maturity, hence, the CGA uses time-steps rather than generations with most of the population being present in succeeding time steps and with breeding producing only a few new children per time step.

3.3.2 Age and Life-span:

In the CGA, for the initial generation, each individual is given a random age which is incremented with each time-step, just like a real individual. So there is a mix of ages within the population. Life-span is the pre-determined death age of an individual. And the life span for the initial population is set based on fitness. When their age exceeds their life-span, the individual is deleted from the population - step 2 in the framework of over-age deletion. Additionally, the concept of accident or illness has been investigated where some individuals die before they reach their allotted life span. Life-spans are based on their fitness comparing to the average fitness of the whole population (there is an example of how to assign life-span in 3.4.1).

3.3.3 Breeding Age, Effective Fitness and Population Creation Rate:

For most life forms, during the early stages of their life, they are not able to breed and so within the CGA individuals are not allowed to breed as a mimic, all individuals have an age at which they reach puberty. Intuitively, this would seem to be an irrelevant feature in a GA but as is shown after (Chapter 4), it can have a significant impact. So, in the CGA only the individuals whose age exceeds breeding age can be selected to be parents.

In the CGA, every individual is given a fitness using a fitness function, as in a typical GA. If the individual enjoys good health and there are no population stresses then the fitness will remain constant throughout life. However, a mechanism called illness parameter is used to reduce the fitness for individuals who are subjected to an accident or disease. Effective fitness is helping to control the ability of individual being selected as parent when reproducing the new offspring. Population creation rate is used to determine the number of children which can be created in a single time step.

It is typically expressed as an equation, viz:
Number of children = Population creation rate × Total population.

Population creation rate is a very important number which makes CGA different from Canonical GA: it is usually a very small number in CGA; sometimes only one or two children per time-step will be born by crossover (and might with mutation afterwards).

Using a small number for population creation rate is because we need to fix a population creation rate to control the total population size, and if it is too big, the size of the population will increase without stop. More details about population controlling will be shown in Chapter 4.2. Moreover, although the population creation rate is not high, i.e. the number of children is not big; it can still find the optimal solution effectively.

3.3.4 Summary of New Parameters in CGA

So, every individual in the CGA has, as well as a fitness parameter, such as in canonical GA, an age, maximum lifespan and effective-fitness.

As described in the framework, the principal differences between the CGA and other GAs can therefore be summarised as the ability of an individual to survive for multiple time steps (there are no generations): delayed reproduction and a population structure in which individuals within the population are not mostly replaced in every iteration and every individual has an age which is either given during the creation of the initial population or for later individuals it is set at zero when they are born and subsequently incremented for each time step.

The population size in CGA is not static and is determined largely by the life span of individuals and the population creation rate of the population. If the parameters are set up correctly, when the population finds a good enough solution, the run can be terminated. On the other hand, if the search is failing, then the population size is gradually reduced to zero during the run or increased without stop, and again, the run terminates. Parameters -- age, life-span -- are also introduced to help controlling population growth in size (Chapter 4.2). For CGA, each individual has its own age and life-span, only when its life-span expires or has accident (Chapter 5.2), it can be deleted from the population. It will not be replaced by any other individuals.

In CGA, there are two fitness-dependent pressures to improve the fitness of the population. Firstly, as usual, the fitter individuals are assigned a greater probability to be selected as parents for crossover. Secondly, since there is only a small number of genetic crossover taking place in any iteration (compared to Canonical GA), the fitter individuals are also assigned a longer life-span so that they can remain in the genetic pool for a longer period, and hence have a greater probability of being selected for crossover at some point. (See explanation for Figure 3.3) The life-span assigned to an individual is thus dependent on its own fitness with respect to the fitness of the rest of the population. The life-span value assigned will be between a preset lower and upper bound. The values for these limiting bounds, relative to the maximum number of time-steps, are important, since the number of time-steps divided by the typical life-span of CGA is an approximate comparison to the number of generations in Canonical GA. The number of time-steps in CGA thus needs to be at least six times more than the typical life-span, in order to allow enough genetic operations to take place in a run.

In every iteration, the population is both decreased through deletion of individuals and increased through crossover. Age-based deletion is simply the removal from the population of individuals whose age exceeds their life-span. The population is then increased through usual genetic operators, but the difference here is that only a relatively small number of children are created, according to the prevailing population creation rate, whereas in Canonical GA, children are necessarily generated to maintain a stable population size. The population creation rate can be proportional to the population size (which is normal in natural genetics), and/or given an over-riding finite number independent of population size (which is useful if the population size gets too small or too big).

More details about population creation rate, effective-fitness and breeding age will be discussed in Chapter 4.

3.4 Different Species Modelling

The theory of Genetic algorithm is based on modelling the genetic evolutionary rules in the natural species. However, even in the real world, some of the species adapt much easily to their surroundings. As a result of making the GA closer to real genetics, CGA introduced several new items, such as population creation rate, age, and life-span etc. Different species have different breeding patterns and life spans. For example, fishes lay huge numbers of eggs, which hatch and produce hundreds of offspring of which only a small portion will survive. A somewhat different behaviour is exhibited by rabbits, which have a relatively short life span, but they can give birth 4 times a year, and with 4-6 offspring being produced each time. In addition, monkeys in many societies only have 1 or 2 children in their whole life; however, their span is typically longer than the former two species.

We have tested three models in the CGA, which correspond to the above species in their reproductive behaviour and life spans. The tests are to determine how various features impact on the performance of the CGA. The tests have been made using two-variable quadratic function and Ackley's function. The results of the tests are given in Table 3.1 and 3.2. The failure rate is the number of times that the GA has not found the correct solution. The time steps are the number of steps to find the correct solution. The number of evaluations gives an indication of the amount of work the GA had to do to find the correct solution, with a low number indicating a lesser amount of work.

As is shown in Table 3.1 below, the comparison is made using two factors:

The first is *failure rate*, which is calculated by out of the 15 runs, how many times the program has not found an acceptable result within 300 time-steps;

The second factor is an *average number of evaluations*, which shows how many children have been created during the search for the best solution. Generally speaking, the fewer, the better. *Number of evaluations* is a measure of the efficiency of the search.

Table 3.1: Results of failure rate and Average number of evaluation by using different species models on Hyperbolic and Ackley function.

Species models	Failure rate	Average Number of evaluation
Two variables Hyperbolic Function		
Fish	0.26	1131
Rabbit	0.33	1396
Monkey	0.06	**170**
Two variables Ackley Function		
Fish	0.4	9181
Rabbit	0.2	1709
Monkey	0.4	**441**

From Table 3.1, it can be seen that Monkey model was overall slightly more successful, but took significantly fewer number of children to achieve this result in both hyperbolic function and Ackley function.

In Table 3.2, details of Ackleys's function test have been listed to show the result for every run and average number of evaluations of all runs and only success runs.

Table 3.2: Comparison result of success time steps and number of evaluations by using different species models for Ackley function.

Run No.	Monkey		Rabbit		Fish	
	Time steps	Number of evaluations	Time steps	Number of evaluations	Time steps	Number of evaluations
1	**4**	**8**	**6**	**12**	**5**	**200**
2	**12**	**24**		7920	**16**	**380**
3	**10**	**20**	**5**	**84**	**4**	**150**
4		1039	**17**	**264**	**8**	**250**
5		835	**19**	**344**		26460
6	**151**	**270**	**14**	**216**	**8**	**190**
7		824	**17**	**276**	**7**	**250**
8	**16**	**32**	**5**	**88**		26250
9		1214	**21**	**328**	**10**	**340**
10	**10**	**20**	**14**	**216**		26100
11		925		7384		25620
12	**6**	**12**		8040	**18**	**770**
13		1267	**7**	**140**		25580
14	**24**	**53**	**7**	**152**	**8**	**320**
15	**33**	**66**	**4**	**72**		5110
Failure rate						
	0.4		0.2		0.4	
Average number of evaluations of all runs						
	441		1709		9181	
Average number of evaluations of success runs only						
	56		183		317	

All of the runs of in each modelling tests, initial population is exactly the same, which means although they start from the same point, but results might be different. The blank time-steps block in Table 3.2 means it has not obtained an acceptable solution before the program being terminated at 300 time-steps. The number of evaluation for that unsuccessful run is the total new individual numbers who has been created until program ends. The two average number of evaluation shows Monkey model is effective even on just successful runs. This indicates that, for the chosen test functions, a lower birth rate and relatively longer life span lead to savings in computational effort. Thus it would seem that keeping genetic material for a number of time steps and having a relatively low death rate has some advantages. Given the better-performed Monkey model, this is the only model used in all of the following tests. Not that, for the rabbit and fish models, two extreme situations occurred. Either success easily obtained or acceptable results can never be shown. By checking details of the individuals, it seems that when two parents create offspring, as they are allowed to create many (several) in one time-step; it results in many of the new offspring being similar to each other. And individuals' life-spans of these two models are comparably shorter. So when these new individuals go into the whole population, older ones were deleted. Therefore, it dilutes the variety of the gene pool. This might be the reason why after certain time-steps, the population are full of similar individuals thus best results are hardly achieved.

3.5 Behaviours of Single CGA

3.5.1 Performance of CGA

Figure 3.2 shows the behaviour of CGA on an optimisation search of a two-variable quadratic function (0 is the final goal) over twenty separate runs.

The function of this optimisation problem is:

$$f(x_1,x_2)=(x_1-9)^2+(x_2-11)^2 . \qquad (1)$$

Ordinary binary coding and tournament selection criterion were used. 28 bit binary code was set in programming for each individual, 14 bits each for parameters x_1 and x_2. The population size was initially set at 40 and life-span values were assigned according to fitness ranking in the population: the minimum and maximum of 30 and 50 time-steps were assigned to the least and the most fit individuals respectively. Life-spans for other individuals were linearly scaled based on their fitness (Fig. 3.2).

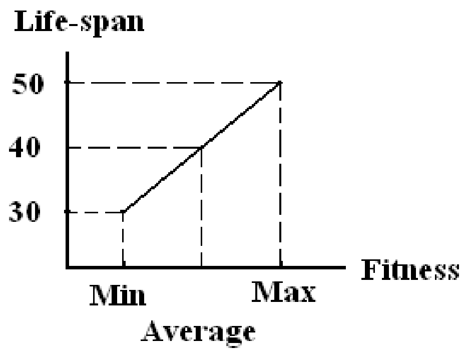


Figure 3.2: Life span evaluation method.

As shown in Fig 3.2, the linear life span evaluation function is the simplest case. The minimum and maximum value is randomly chosen as an example to show no matter how fitted one individual being evaluated in each time step, it may vary in real programming. The rule is: individual with average fitness is set with a life-span of 40;.The better the fitness, the bigger the life-span. The best fitted individual will have a life-span of 50. On the contrary, the worst fitted one can only have a decreased life-span of 30. In this case, a maximum age of 50 was allocated to each individual in this initial population. At the start of the first iteration, only a few individuals were deleted due to their long age. But when the initial population creation rate was set at 5% of the population whenever it is below 80 (and at 2.5% otherwise), the population will increase to over 80 in about the first 50 time-steps (Fig. 3.3).

Fig. 3.3 shows the size and mean fitness of the population, and the fitness of the best fit individual, over the time-steps. The behaviour and pattern in each of these were similar for the 20 runs. Generally, where the population creation rate (explained in previous chapter 3.3.3, and discussed in following chapter) is around or just under 2.5%, the population size eventually stabilises. As the population creation rate increases (e.g. 5%), then the population would steadily and continuously increase. It can be seen in Fig. 3.3 that once the population has stabilised; it hovers generally around 80. Although it appears in Fig. 3.3 that the scatter in the data decreases with time (e.g. the scatter was smaller at time-step 900 than at time-step 300), this

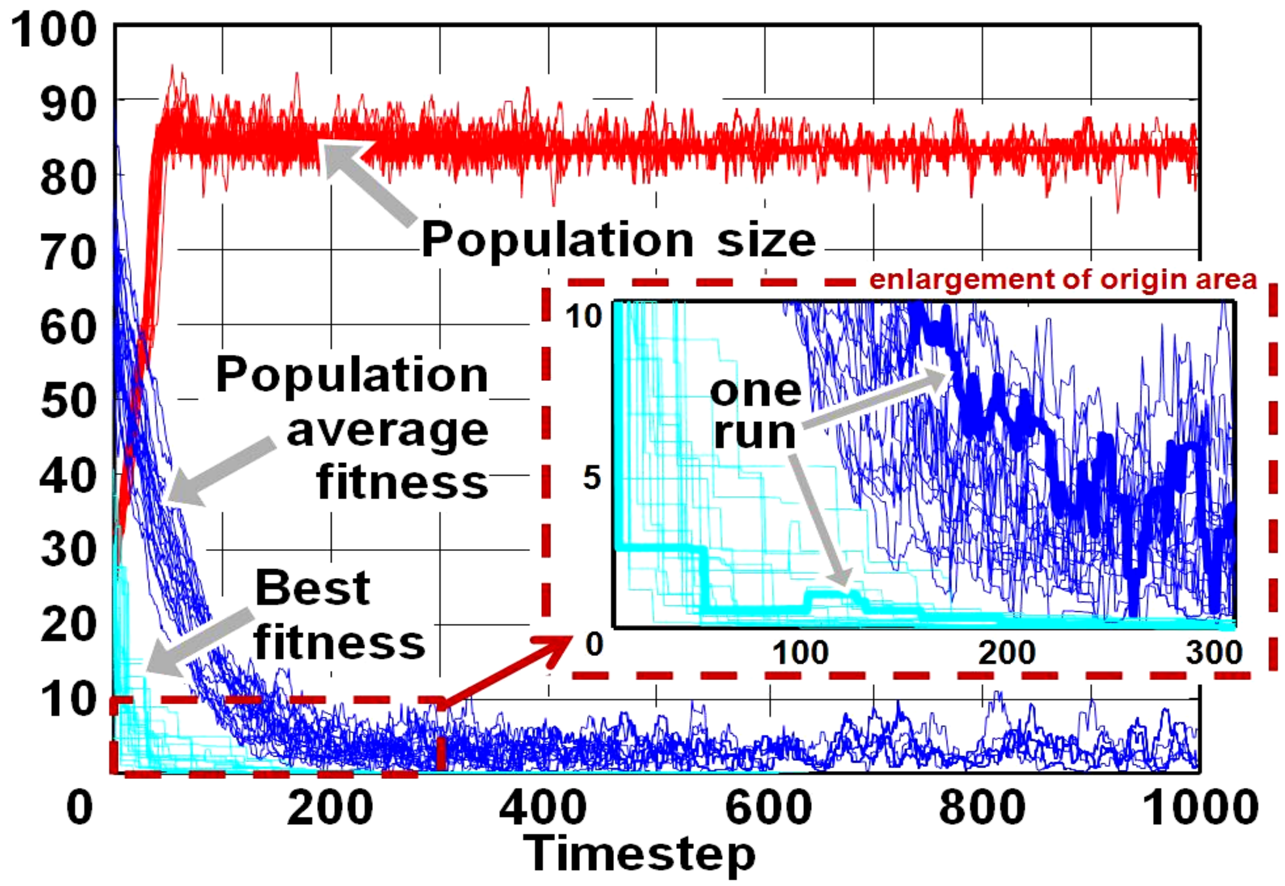


Figure 3.3: Characteristics of CGA over 20 runs.

apparent "convergence" is due to the fact that only five of the 20 runs shown in Fig. 3.3 continued for 1000 time-steps: the other 15 terminated earlier because the fitness criterion had been achieved.

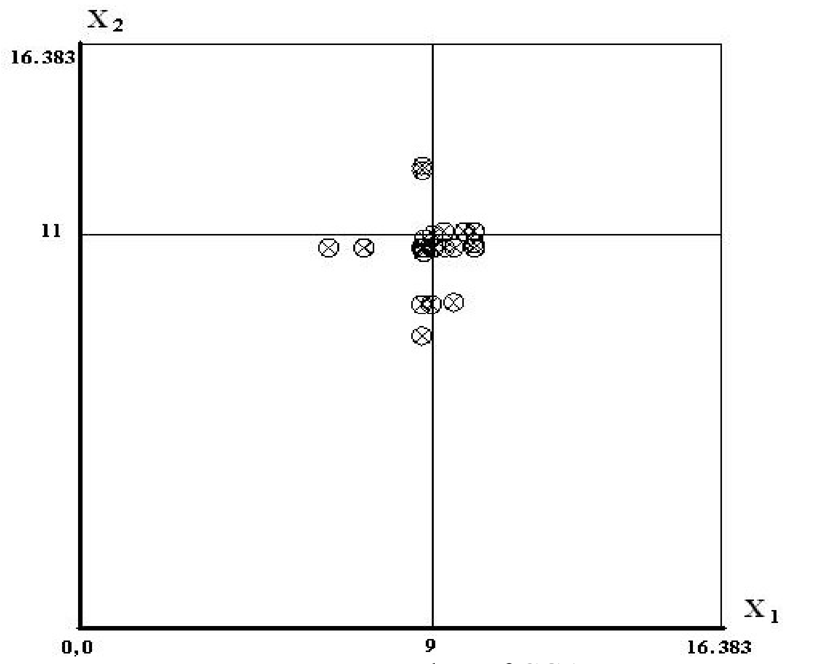
It can also be seen in Fig. 3.3 that, at least for the function in eqn.1 $f(x_1,x_2)=(x_1-9)^2+(x_2-11)^2$, the CGA exhibit convergent behaviour with a progressive improvement in the population fitness. The initial population typically began with an average fitness of around 50 to 70, and there was then an exponential-like sharp early improvement to fitness of around 5 by time-step 50-100, with a gentle gradual improvement thereafter. Since the maximum lifespan in these CGA runs was 50, in the period of time-step 50-100 of CGA population will basically change to all new offspring which would be equivalent to the first two generations in Canonical GA (i.e. the two generation after the initial randomly generated population) (based on the evaluation of children creation numbers, assuming crossover rate in Canonical GA is 0.5, and population creation rate in CGA is 0.03/0.05). In light of this, the rapid population improvement by time-step 100 is thus quite remarkable.

The fitness of the best individual also improves very quickly, as shown by the enlargement in Fig. 3.3 of the area around the origin. A typical run (is been highlighted in thicker lines to more clearly) show its behaviour. It can be seen that, for successive time-steps, the average fitness of a population fluctuates, and the fitness of the best individual is more stable but also it fluctuates due to the death of the fittest individual (there is no elitism being used). This is a typical feature of the CGA where only two children are produced per time-step, so one would not expect a better individual to emerge with every time-step, or even every few time-steps. Indeed, a best-fitness individual can remain the so throughout its lifespan, and as already mentioned, when it is eventually deleted due to old age, the fitness of the fittest individual can decrease as can be seen at around time-step 100.

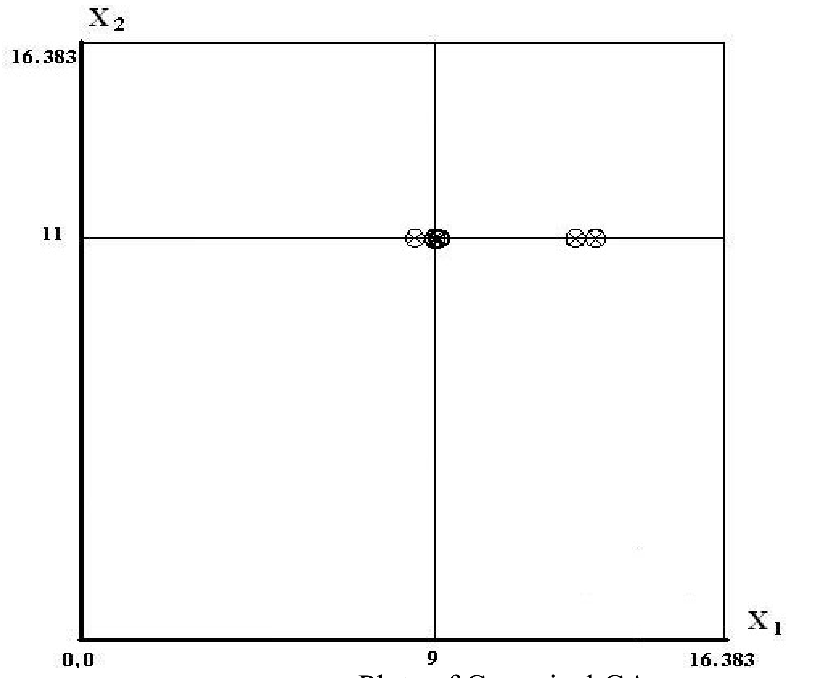
There are two fitness-dependent pressures to improve the fitness of the population in CGA, these are:

1. The breeding election process and
2. Life-span.

The parent selection mechanism (tournament selection) is the same as that which is used in many ordinary GA programming, whereas life-span/effective fitness is a particular feature of the CGA. The lifespan assigned to an individual is thus dependent on its own fitness with respect to age (Fig. 3.2). In a typical GA, weak individuals tend to be quickly replaced by new off-spring, so convergence tends to happen relatively quickly. However, as is often stated, there is a possibility that the weaker individuals contain some good genetic material. So that giving a reasonable life-span, all the individuals can enhance the chances of selecting poorer individuals which may possess a reasonable amount of advantaged genes. This is likely to slow



Plots of CGA



Plots of Canonical GA

Figure 3.4: Plots of fitness when the fittest individuals are close to the optimum.

the convergence process while a good solution is still achievable. To show how the search process is different for the two types of GA, in Fig. 3.4, gives a plot of all the individuals in the search space of a canonical GA, when the search is close to the optimum. As can be seen, all the individuals are crowded around the one solution point. However, for the CGA there is still a significant spread in the population so diversity is being maintained for longer and yet the fittest solution is very close to the optimum. The all area of x_1 and x_2 is the initial searching area. And the cross at (9, 11) is the final goal.

The modelling of the real world phenomenon where stronger and healthier individuals tend to live and the dynamic re-assignment of the life-span for each individual at every time step (since the fitness of the population evolves, and hence the ranking of an individual within the population will change with each time-step), this can be a computationally wasteful exercise, because it is unlikely to bring much change. Therefore, in the results presented in Fig.3.4, life-span is evaluated only once for each individual, either at the initial random population setting or at birth in later operation, and the average fitness of the whole population at that time is used as the benchmark; higher fitness one will get longer life-span, while lower one's life-span will be reduced (Fig. 3.2). Therefore, in the current work, the lifespan has been static for each individual, either at birth or at the initiation of the run where the life span itself is kept constant during the creation of the initial random population.

3.5.2 Three Test Functions

In order to further compare the performance of a Canonical GA and the CGA, three test functions have been selected, namely:

1. Quadratic hyperbolic function,

$$f(x_1,x_2)=(x_1-9)^2+(x_2-11)^2$$

2. De Jong's fifth test function

$$\begin{cases} \max f_4(x_1,x_2)=0.002+\sum\limits_{j=1}^{25}\dfrac{1}{j+\sum\limits_{i=1}^{2}\left(x_i-a_{ij}\right)^6} \\ -65.536\le x_1,x_2\le 65.536 \\ \left[a_{ij}\right]_{2\times 25}=\begin{bmatrix}-32,-16,0,16,32,-32,-16,\cdots,0,16,32\\ -32,-32,-32,-32,-32,-16,\cdots 32,32\end{bmatrix}\end{cases}$$

3. Ackley's function

$$f(x_1,x_2)=e^{-0.2}\cdot\sqrt{x_1^2+x_2^2}+3(\cos 2x_1+\sin 2x_2)$$

All of them are with two variables.

The following results focus on the comparison of the efficiency and effectiveness of the two methods. For all the results, both methods started using exactly the same initial population for each function, which were created by choosing the 40 weakest from a total of 60 randomly created individuals in Hyperbolic and De Jong's functions and 80 out of 120 in Ackley's function. This was done to make the problem reasonably challenging. The crossover rate used in the canonical GA was 60%, with the 24/40 poorer individuals being replaced and the top 16/40 being passed through to the next population. For the CGA, the population creation rate was set as 5% when the population size was below 80, above which, it was reduced to 2.5% in Hyperbolic and De Jong's function; 3% and 1.5% judging by population size of 160 in Ackley's function. So typically, fewer than four individuals were created every time-step, and those children were added to the total population but did not replace any former individual. There is only one way of removing an individual from the population and this is when its life-span is achieved. The mutation rate was the same for both methods at 2%.

Both types of GA have been programmed to run on the same computer using code written in MS Visual C++ (thanks to the help from books written by Horton, Meyers, Michalewicz, Raphael and Schildt) with every effort being made to make the code identical where possible. The three test functions used and the respective results are presented below. In each case, as well as the optimum solution, an acceptable solution is given, this being a solution which is deemed close enough to the optimum for convergence to have effectively occurred. The details and discussion of comparison results will be shown in section 3.5.3.

3.5.2.1 Quadratic hyperbolic function

The first test function is a hyperbolic function with two variables. In Table 3.3 it shows the details for the quadratic function and its solution information:

Table 3.3: Two variable hyperbolic function and solution details

Equation	$f(x_1, x_2) = (x_1 - 9)^2 + (x_2 - 11)^2$ (2).
Optimum result	f (9, 11) = 0
Acceptable solution	f < 0.4.

Several parameters have been used in the two types of GAs. Lists are shown and compared in Table 3.4:

Table 3.4: Parameters for hyperbolic function used in Canonical GA and CGA

	Canonical GA	CGA
Coding type	Binary code. 14 bits for each parameter of x_1 and x_2. 28 bits string for every individual	
Initial population size	40	
Crossover type	Single point crossover	
Crossover rate (Canonical) Population creation rate (CGA)	60%	Population <80, 5% Population ≥80, 2.5%
Mutation rate	2%	2%
Generation limit (Canonical) Time steps limit (CGA)	50	300
Deletion method	Replaced by new individuals	Life span exceeding
Special parameters		Age range: 30-50 Initial ages are random chosen Effective fitness type: box Accident rate: 1%

Both Canonical GA and CGA started from the same initial population, Table 3.5 below shows the best and the worst 3 individuals' two values with its own fitness and age:

Table 3.5: The initial population for hyperbolic function used in Canonical GA and CGA (the best 3 and the worst 3 individuals are shown)

x_1	x_2	fitness	age
1.16	0.285	176.277	42
0.608	0.843	173.59	45
16.186	0.748	156.742	37
⋮	⋮	⋮	⋮
2.787	11.712	39.1083	17
4.103	7.223	38.2463	15
14.823	9.14	37.3669	49

It can be seen that the initial fitness levels were all far away from the target 0. The results for the application of this test function to both types of GA are presented in Figs. 3.5 & 3.6. These clearly show that there are some significant differences in performance between the two variations of the GA.

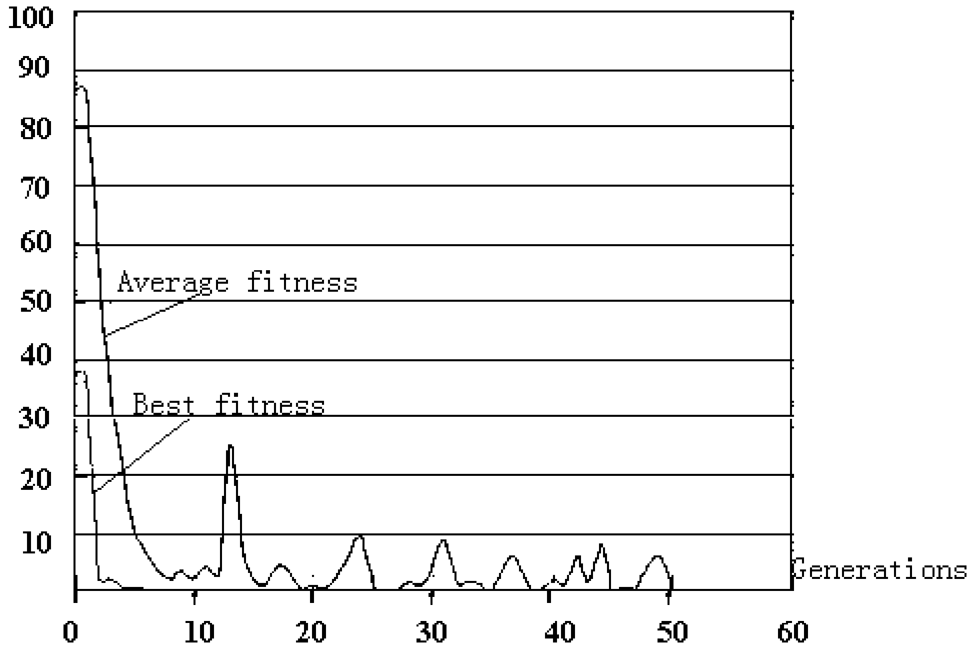


Figure 3.5: Canonical GA with hyperbolic function

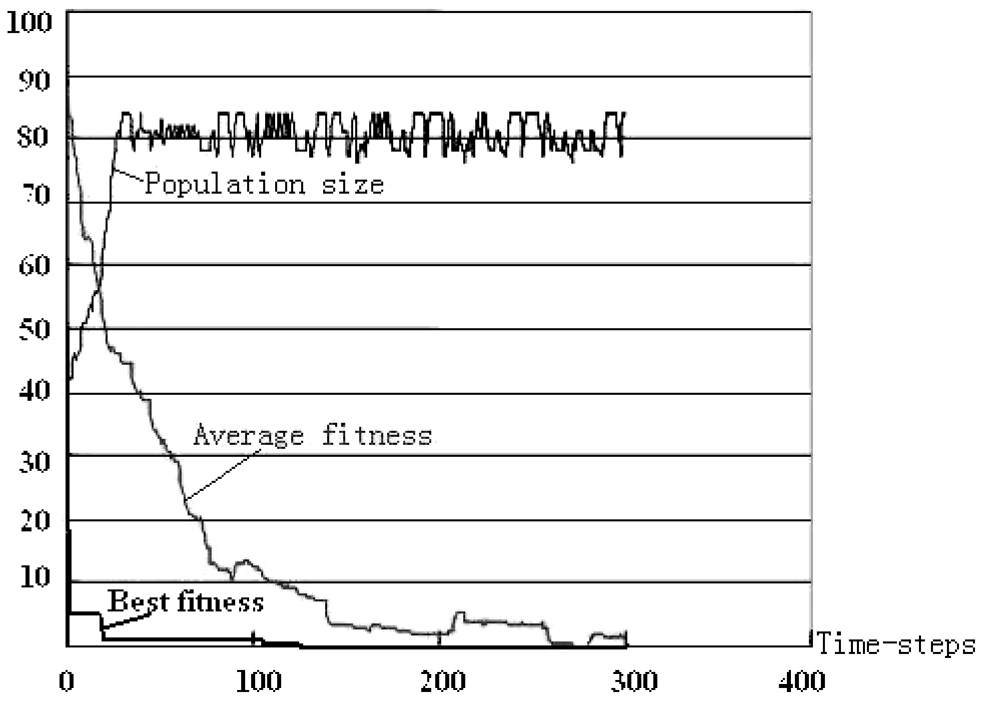


Figure 3.6: CGA with hyperbolic function.

In both figures, the best fitness and average fitness show significant improvements during the initial stages, with the canonical GA appearing to perform better. However, the two plots have different X axes. For the canonical GA the X axis is for generations, whereas for the CGA it is time steps. In a CGA time step, only a few individuals are created and deleted, and fitness evaluation is only needed for the new individuals. Therefore, in terms of computational effort, fifty generations of the Canonical GA are roughly equivalent to 300 time steps of the CGA. The population size is only shown in the plot for the CGA because only in the CGA is a dynamic population size allowed. For the canonical GA, the population size is constant at forty.

As shown in Figs 3.5 & 3.6, the average fitness of the CGA is much less variable than that of the Canonical GA. The average fitness of the canonical GA exhibits some significant fluctuations through the generations, the main reason being 60% of the population is changed in every generation, and this significant percentage of new individuals obviously affects the average fitness. Another reason is in the Canonical GA, the population convergences very quickly (see Fig. 3.4), all the individuals are very similar, that may results in no obvious improvement by crossover, so the main improvements in the offspring are created by mutation. On the other hand, for the CGA, in each time step, only two to four children are created by and less than five individuals are deleted for exceeding their life-spans. Considering the population size is around 80, the influence from new population members and death on the average fitness is small, and hence the fitness curve is smoother.

3.5.2.2 De Jong's fourth test function

This is the next test function; one of De Jong's functions has been chosen to test. Table 3.6 gives the information of this function.

Table 3.6: Two variables De Jong's function (1975) and solution details.

Equation	$\begin{cases} \max f_4(x_1, x_2) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6} \\ -65.536 \le x_1, x_2 \le 65.536 \\ [a_{ij}]_{2\times 25} = \begin{bmatrix} -32, -16, 0, 16, 32, -32, -16, \cdots, 0, 16, 32 \\ -32, -32, -32, -32, -32, -16, \cdots 32, 32 \end{bmatrix} \end{cases}$ (3).
Final result	f (-32, -32) = 1
Accept solution	f > 0.98.

Table 3.7 gives the parameters used for the tests on De Jong's function.

Table 3.7: Parameters for De Jong's function used in Canonical GA and CGA.

	Canonical GA	CGA
Coding type	Binary code. 15 bits for each parameter of x_1 and x_2. 30 bits string for every individual	
Initial population size	40	
Crossover type	Single point crossover	
Crossover rate (Canonical) Population creation rate (CGA)	60%	Population <80, 5% Population ≥80, 2.5%
Mutation rate	2%	2%
Generation limit (Canonical) Time steps limit (CGA)	300	500
Deletion method	Replaced by new individuals	Life span exceeding
Special parameters		Age range: 30-50 Initial ages are random chosen Effective fitness type: box Accident rate: 1%

Three best and three worst fitness individuals from the initial population for De Jong's function are shown in Table 3.8 with its own fitness and age.

Table 3.8: The initial population for De Jong's function used in Canonical GA and CGA (the best 3 and the worst 3 individuals are shown)

x_1	x_2	fitness	age
-61.202	55.368	0.002	20
-53.775	61.128	0.002	39
38.844	-63.976	0.002	44
⋮	⋮	⋮	⋮
27.721	40.76	0.00200253	26
-40.439	27.608	0.00200309	44
-7.536	-39.488	0.00200465	9

Same again as hyperbolic function, the initial population for De Jong's function is also the worst 40 from random chosen 80 individuals. All of them are far away from the real solution.

Figure 3.7 and 3.8 shows the performance of De Jong's function on Canonical GA and CGA separately.

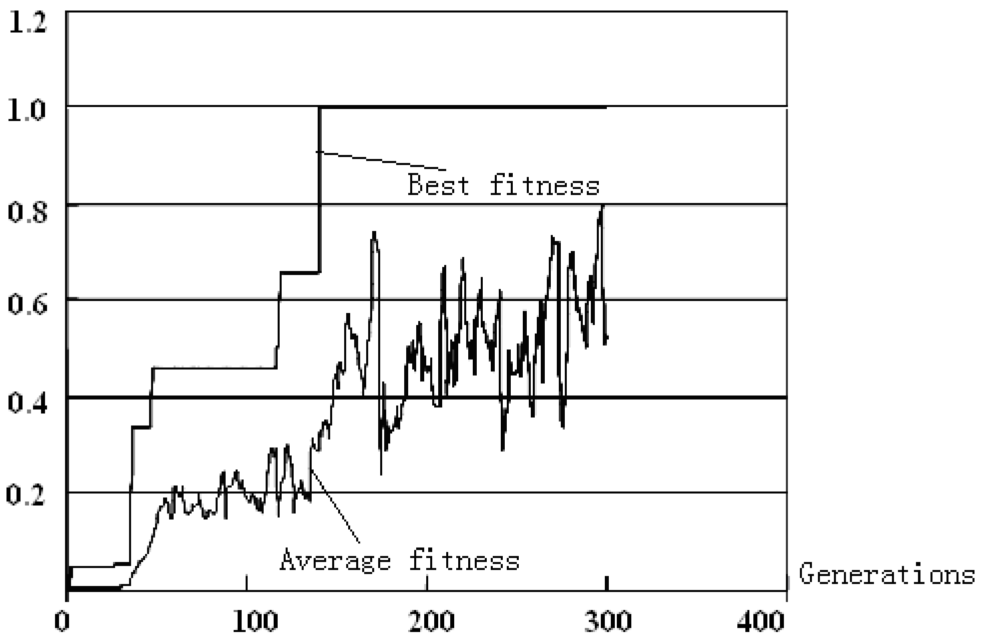


Figure 3.7: Canonical GA with De Jong's function

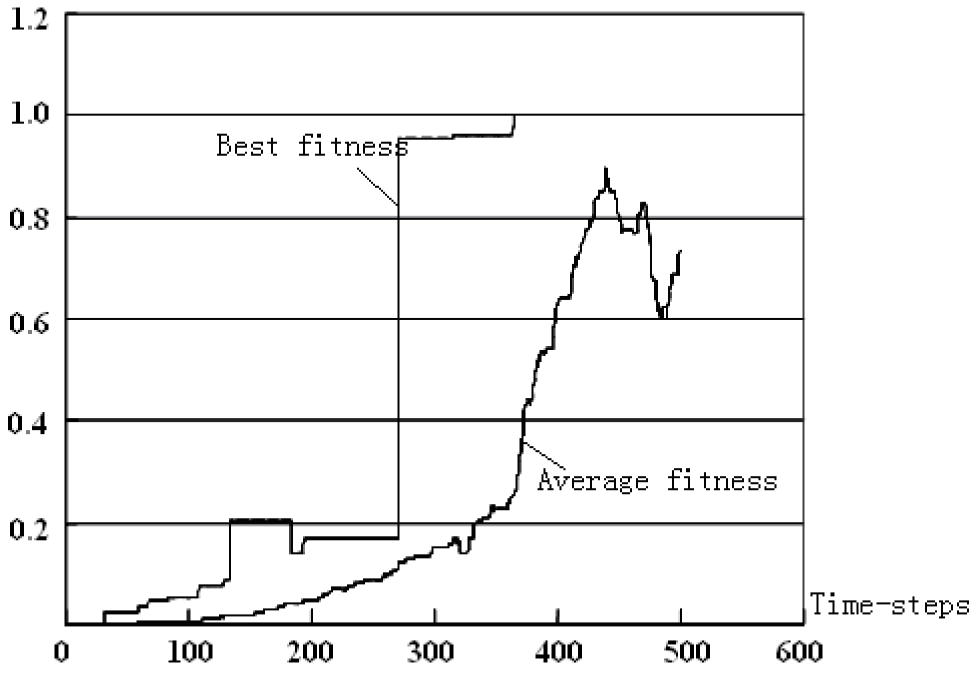


Figure 3.8: CGA with De Jong's function

Figs. 3.7 and 3.8 give the results for De Jong's function with the former showing the performance of the canonical GA and the latter with the CGA. The x axis of the former is for generations and for the latter, time steps; so the amount of work involved for 400 generations is substantially more than for 600 time steps. In Fig. 3.7, the final best fitness solution from Canonical GA comes at about the 140 generations. And in Fig. 3.8, the final solution from CGA comes around 360 time-steps.

It can be seen from Figs. 3.7 and 3.8, the average fitness of the two methods figures quite different shapes. The average individual during the progress in Fig. 3.7 keeps changing drastic between generations. The reason for this is because in every new generation, there are always some new fitness individuals, created by crossover and mutation, replaced part of the previous population. The new ones might be worse than former ones, so the average fitness have been reduced. On the contrary, average fitness curve is much smoother. The reason is there is much less population changes in CGA. And all new individual is added to the whole population not replacing any of them. There are still some deletions by exceeding life-span or accident risk but still less compared to the Canonical GA.

Another important observation from Fig. 3.8 is the drop in the best fitness at about 180 time-steps. Such behaviour can happen in CGA but the same is not possible in canonical GA when using elitism.

3.5.2.3 Ackley's test function

The third function to test is Ackley's function (1987).

Function details are shown in Table 3.9.

Table 3.9: Two variables Ackley's function and solution details.

Equation	$f(x_1, x_2) = e^{-0.2} \cdot \sqrt{x_1^2 + x_2^2} + 3(\cos 2x_1 + \sin 2x_2)$ (4)
Final result	f (1.5096, -0.7548) = -4.5901 f (-1.5096, -0.7548) = -4.5901.
Accept solution	f < -4.179466

Parameters used on Ackley's function have been compared in Table 3.10.

The initial population used on Ackley's function is 80. The best five and worst five from the initial population are shown in Table 3.11 with its own fitness and age.

Table 3.10: Parameters for Ackley's function used in Canonical GA and CGA.

	Canonical GA	CGA
Coding type	Binary code. 26 bits for each parameter of x_1 and x_2. 54 bits string for every individual	
Initial population size	80	
Crossover type	Single point crossover	
Crossover rate (Canonical) Population creation rate (CGA)	60%	Population <160, 3% Population ≥160, 1%
Mutation rate	5%	2%
Generation limit (Canonical) Time steps limit (CGA)	300	500
Deletion method	Replaced by new individuals	Life span exceeding
Special parameters		Age range: 50-150 Initial ages are random chosen Effective fitness type: box Accident rate: 1%

Table 3.11: The initial population for Ackley's function used in Canonical GA and CGA (the best 5 and the worst 5 individuals are shown)

x_1	x_2	fitness	age
-6.40122	-5.12326	11.8265	12
-6.14135	-2.4903	11.1986	20
-3.36298	-5.44451	10.9331	100
-3.4027	6.57897	10.3373	9
3.55048	3.78686	9.1845	83
⋮	⋮	⋮	⋮
2.30763 -	0.342879	0.280655	172
1.82824	2.95866	0.83682	83
1.3106	5.54501	-0.924539	88
-4.88734	-.0672618	-1.70308	87
1.5775	3.62062	-.2.2205	131

Figs. 3.9 and 3.10 give the results for De Jong's function with the former showing the performance of the canonical GA and the latter with the CGA.

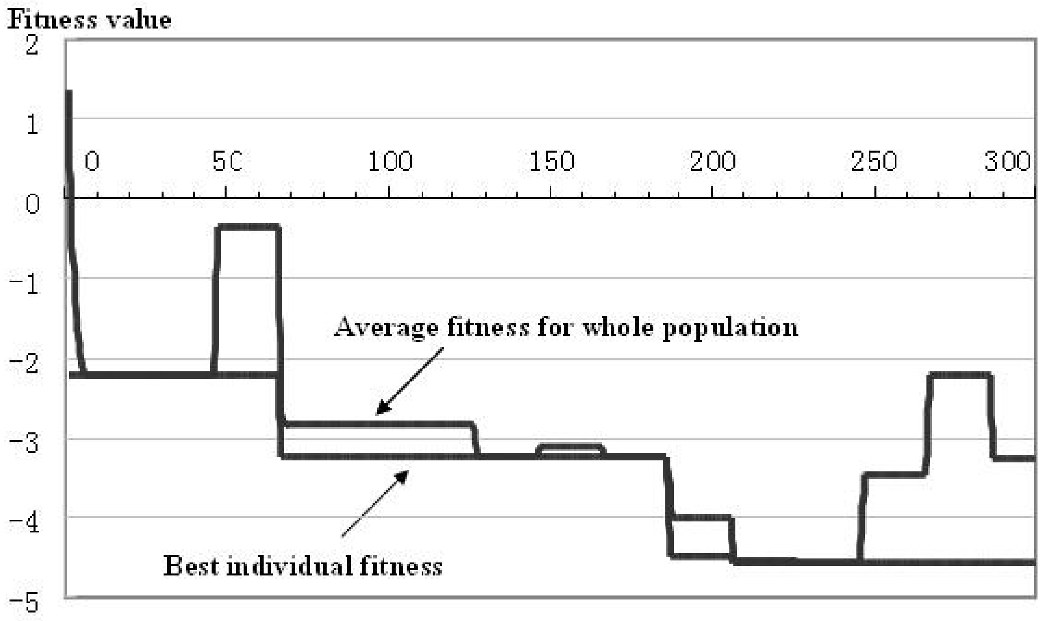


Figure 3.9: Single population Canonical GA for Ackley function

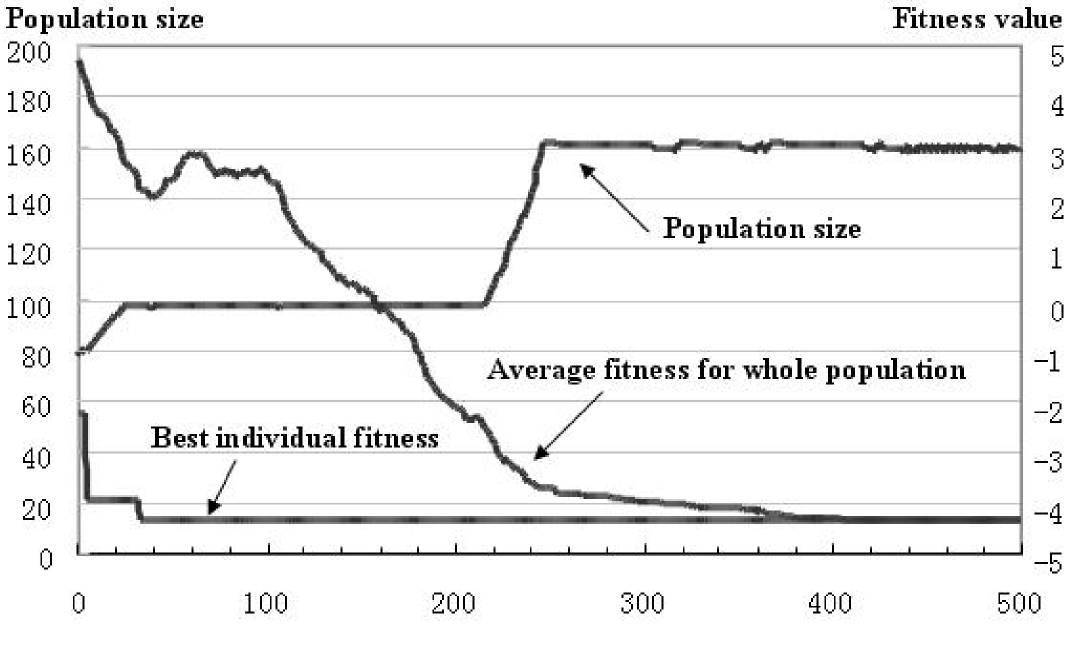


Figure 3.10: Single population CGA for Ackley function

One of the pitfalls of comparing algorithms and their efficiency / effectiveness is that it is possible to improve the performance by reconfiguring the parameters and operators and so an absolute comparison cannot be made. In this work for the Canonical GA, single point crossover was tried first, but the algorithm tended to converge fairy quickly on sub-optimal solutions. It was found that multiple point crossovers gave better performance and so this has been employed to obtain the above results.

3.5.3 Performance Comparison between Single CGA and Canonical GA

Except the core different of the method ideas, all other conditions were the same in Canonical GA and CGA. For every function, each method has been tested for 15 runs respectively. From the results for the above three functions, it is possible to make a comparison between the Canonical GA and the CGA, give the above provisos about the comparisons not being absolute and the limitations expressed in the No Free Lunch Theorem (Wolpert & McReady 1997).

As is shown in Table 3.12 below, the comparison is made using two factors:

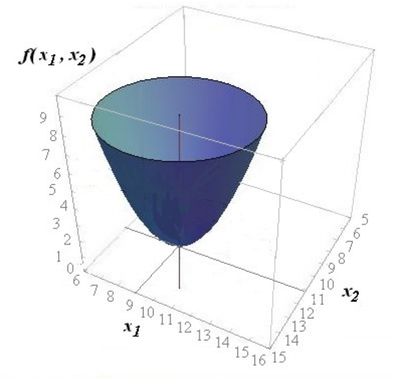
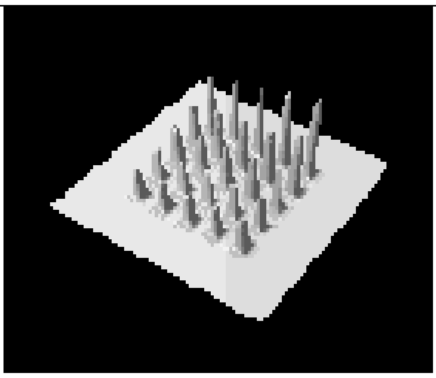
1. *Failure rate*, which means out of the 15 runs, how many times the program not found an acceptable result within 300 time-steps (CGA) or 50 generations (Canonical GA. The *failure rate* is therefore a measure of the performance of the search);

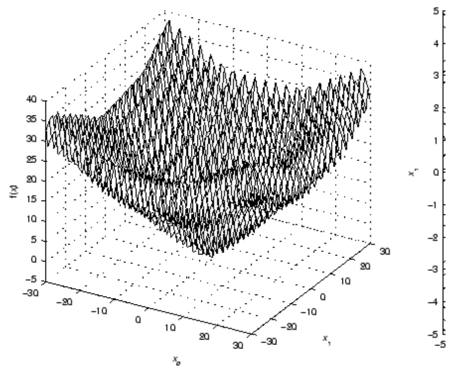
2. *Number of evaluations*, which shows how many children have been created during the search for the best solution. Generally speaking, the fewer, the better. *Number of evaluations* is a measure of the efficiency of the search.

The reason why 300 times-steps in CGA equal to 50 generation in Canonical GA is that: in Canonical GA, crossover rate is 60%, so there are 24 (if the initial population is 40) new individuals in every generation. But in CGA, maximum four new individuals can be created in each time-step under the control of population creation rate of 5% with population 80. In this case, 300 time-steps and 50 generations are equal.

As shown in Table 3.12, these three functions have been chosen be because of their distinctly different shapes. The test summary of the *failure rate* and *number of evaluations* are listed below.

Table 3.12: Performance comparison between Canonical GA and CGA (Hyperbolic, De Jong's and Ackley test functions).

	Canonical GA	CGA
Hyperbolic		
Failure rate	0.4	0.06
Number of evaluations	173	**170**
De Jong's		
Failure rate	0.6	0.53
Number of evaluations	2608	**577**

Ackley		
Failure rate	0.73	0.4
Number of evaluations	12614	**441**

It can be seen from Table 3.13 that the failure rates of the CGA and canonical GA are similar, with the CGA having the edge. For hyperbolic function, the results of two methods are not much different in *number of evaluation* but a slightly better in *failure rate*. But for other two functions, the significant difference is in the average number of evaluations per successful run, in other words the efficiency.

For De Jong’s function, the number of evaluation required by the CGA is almost a quarter of those of the Canonical GA and for Ackley’s, it is around 3.5%. This indicates that on the simple function the canonical is similar to the CGA but on the more complex functions, the CGA does better. For the chosen test functions, the Cardiff GA is more efficient than the Canonical GA. Work using other test functions indicates that these results are typical and in no instance has the CGA been found to be less efficient than the Canonical GA. Cardiff GA can reach the required answer more quickly and with less computational effort.

Take the details from Ackley's function as an example.

As the result of Canonical GA, only 4 times are successful in the 15 runs, which are run 1, 4, 7 and 11. The successful generations are:

110 for 1st run,

227 for 4th run,

119 for 7th run

and 186 for 11th run.

The failure rate is 0.73. The average number of children been produced when the good result is obtained is 12614.

Table 3.13: In CGA, time step numbers and number of evaluations when acceptable results appeared are:

Run No.	Time steps	Number of evaluations
1	**4**	**8**
2	**12**	**24**
3	**10**	**20**
4		1039
5		835
6	**151**	**270**
7		824
8	**16**	**32**
9		1214
10	**10**	**20**
11		925
12	**6**	**12**
13		1267
14	**24**	**53**
15	**33**	**66**

As is shown in Table 3.13, the failure rate for the CGA with Ackley's function is 0.4, which is considerably performed better than the canonical GA. Also the average number of children which have been required to produce these results is 441, compared to 12614 for the canonical GA. This indicates that, for certain types of problem (bearing in mind the No Free Lunch Theorem, introducing the features contained within the CGA can be beneficial.

3.6 Chapter Summary

This chapter briefly introduces the idea of single population CGA. This is a genetic algorithm mimicking natural population. Like age, life span, population creation rate, etc., many familiar names have been used in the CGA modelling program. By using these parameters, a new framework of CGA was built. By giving different values to the new parameters, a variety of species can be mimicked. The result shows a longer life span and lower population creation rate as monkey model has a better performance in some specific function. Using the monkey model, after comparing three different test functions, current tests results show CGA requires fewer genetic operations and is computationally faster than Canonical GA overall when it does locate the correct answer. We could say that the newly proposed CGA brings an improvement on giving more diversity on the searching space in that it offers a new sort of framework for GAs which is much closer to the real world.

Chapter 4 Element Effects in Single CGA

4.1Chapter Introduction

After describing the framework of CGA and the performance comparison between Canonical GA and CGA in three test functions, a deeper question arises: which affects the performance of CGA. Since several new parameters have been introduced into this new method, the influence of these items is worth testing and discussing.

In this part, population creation rate, life-span, effective-fitness and breeding-age have been tested with the two-variable hyperbolic function. At the end of this chapter, there is also a tracking of an individual’s family tree which shows how the best solution is created and what would be the required effect to get it.

4.2 Elements in Population-size Control

Having established the superior performance of the Monkey model, the next step is to look in more detail at the features of life span, the age at which puberty is reached and fertility after puberty is reached. Intuition suggests that keeping mature individuals in the population for longer than is absolutely necessary is a waste, although the results in the previous section indicate there is real benefit to be gained from having individuals with relatively long life spans.

To answer these questions, various tests have been undertaken and these are discussed in the following sections.

Different from Canonical GA, CGA has a changeable population size during the time-steps. How to control the population affects the performance of CGA. Because non-stop increasing population will let the program stuck (i.e. Numbers of population will be increased dramatically in a way not linear but geometrically within short period of time. But computer cannot handle those massive operations and it often lead to break down).On the other hand, the population terminated too early may cause the failure of solution searching.

Fig. 4.1 illustrates the impact that the birth rate generates on the population size. The function for testing was two-variable Hyperbolic function. To compare the birth-rate influence, all the other parameters were kept the same (initial population size is 40;

effective fitness is box type (will be explained in next section 4.3.1), breeding age is between 25 and 50 and maximum life span is 50). The only deletion during the time-step is life-span exceeding deletion.

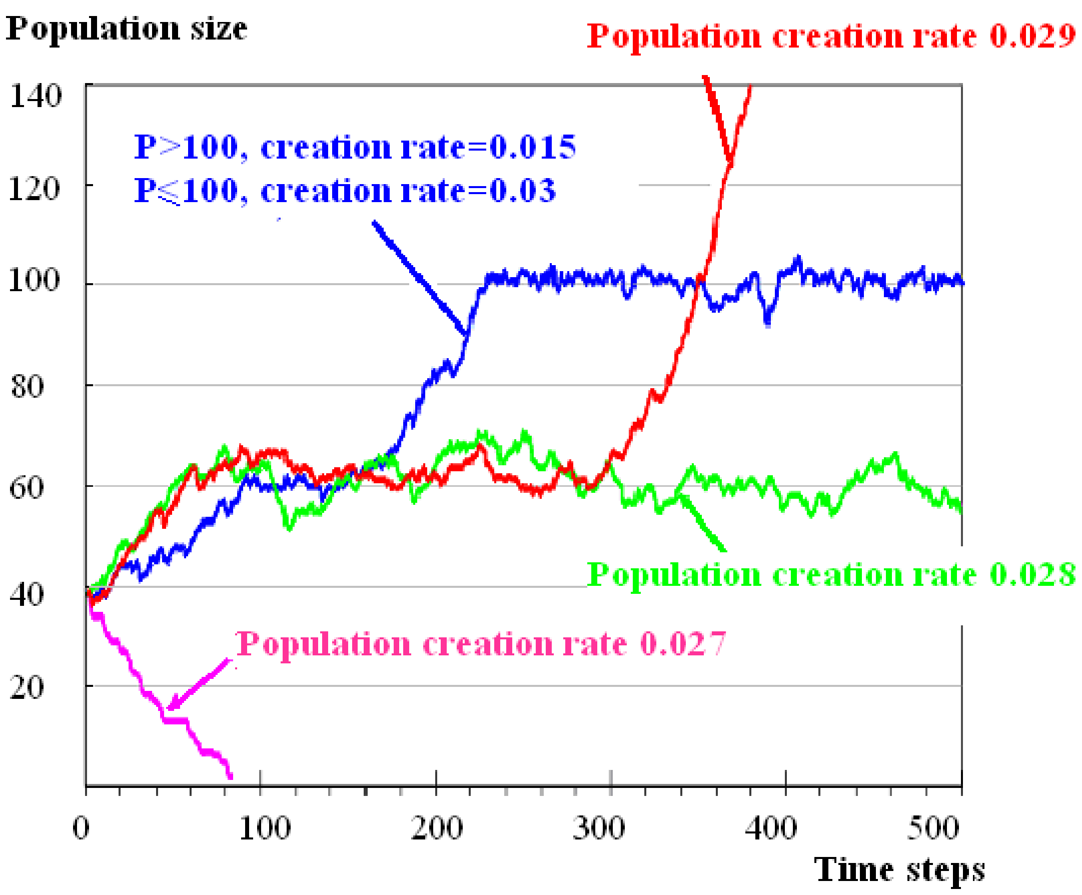


Figure 4.1: Population sizes under different Population creation rates (Hyperbolic function).

In Fig. 4.1, four curves represent the four population changes along with time-steps under different population creation rates. Four population creation rates have been tested: high, low, middle, and a population creation rate according to population size.

They are 0.029 for the high rate, 0.027 for the low rate, 0.028 for the middle rate and a changeable rate control by population size (when the population is less than 100, population creation rate will be 0.03; once it is over 100, 0.015 will be used).

All tests start from the same initial population with the same parameter except population creation rate. Large different numbers have been tested as well with the same function. But those three sets of numbers been selected in Fig 4.1 indicate that a small difference in population creation rate causes significant differences, which means even close numbers make the population very sensitive. The results presented for the population creation rate 0.028 are not always stable as what shown in Fig. 4.1. The reason is the individual been created by crossover and mutation is not exactly the same every time. When the better fitness individual comes at early time-steps, it will remain in the population and create more children for a longer time by have a longer life-span. So it will help population grow. If the new individuals are not good enough, their life-spans will be shorter in result of poorer fitness. The population size will drop down by cutting those off earlier. In this case, the population may reduce to extinction. It is changeable between increase non-stops and extinct early. For the above reasons, the population creation rate used in all the other tests is variable according to population size as this is the only way to keep things reasonably stable. In the tests, it is using a higher rate to produce more children and let population grow, .When it reaches the population limit, lower rate will be used to reduce the new-born speed and let population decrease by life-span exceeding deletion.

4.3 Effective Fitness and Breeding Age

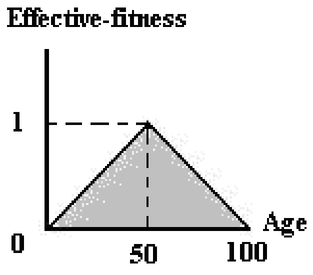
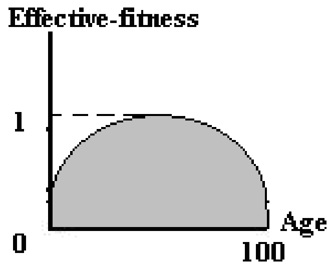
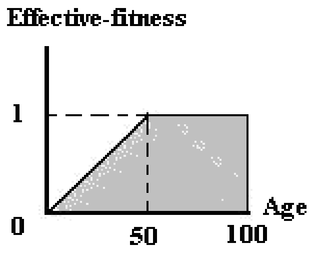
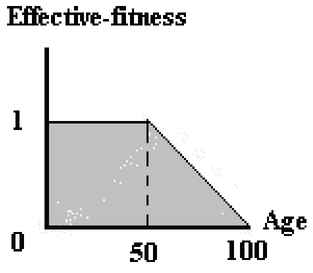
Every species has a span of time during which mature individuals are fertile. In monkey species, this is roughly from the ages of 15 to 45 but the level of fertility is not constant during this time. Assume the CGA is to incorporate more of the features of real genetics, thus there should be some variation in the level of fertility for each individual.

The way that this has been implemented in the CGA is to introduce a feature called effective fitness, where the raw fitness is modified with a function which has been investigated in Table 4.1.

4.3.1 Shapes of Effective Fitness

According to the explanation of effective fitness, ability for individual chosen as parent will be determined by both effective-fitness and its own age. It can be different shapes for effective-fitness in order to enable comparison. The shapes in the Table 4.1 give a graphical representation of the effective fitness with this being a function with a range between zero and one which modifies the basic fitness by multiplication. In the examples shown the individuals have a maximum life span of 100 time-steps and the function applies over the lifespan.

Table 4.1: Comparison of different Effective-fitness (Hyperbolic function).

		Shape (Effective-fitness vs. Age)	Failure rate	Number of evaluations
Triangle			0.06	124
Hyperbolic			0.06	133
Echelon	1		0.93	121
	2		0.06	121

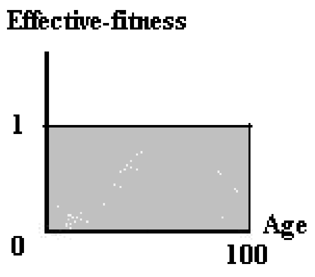
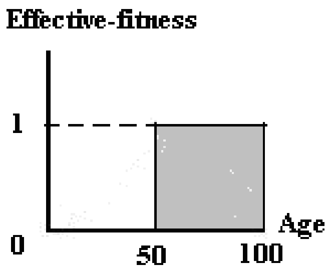
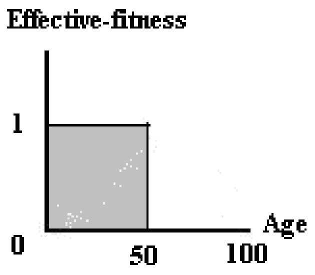
	1		0.06	129
Box	2		0.06	**90**
	3		**0.33**	**263**

Table 4.1, shows that four different types of function have been investigated with variations for some of the functions in terms of the time span to which they apply. As with the previous example, the efficiency and effectiveness is demonstrated using two measures of number of evaluations and *failure rate*. Efficiency is *number of evaluation*, effectiveness is *failure rate*. *Failure rate* counted the number of unsuccessful runs from a total of 15, and success indicates the algorithm located the acceptable solution within limited time steps. *Number of evaluation*, as before, is the

number of offspring produced by the time an acceptable solution is located. In Table 4.1, it is the average of the 15 runs: with, for the unsuccessful runs all 500 time steps being counted.

The most successful in terms of effort is “Box 2” which achieved success on every run with an average of only 90 for the 15 runs. It is interesting to compare this with “Box 3” which is of the same shape but allows early breeding. This shows quite clearly that within the CGA, delayed breeding is a benefit. One presumes that this is because it prevents the breeding of younger individuals with older, less fit members of the population. This hypothesis is supported by a comparison of the performance achieved with the other functions.

4.3.2 Starting Age of Breeding and Best Breeding Age Range

If delayed breeding is a useful feature, the question arises as to what extent it should be delayed. The above example is for 50 time steps but is it the best choice? Another choice is for how long an individual should remain fertile. Again the above examples are for a maximum of 100 time steps, but is this the optimum value? Figure 4.2 gives the results of tests on these two parameters. All other parameters and functions used are the same as shown in Tables 3.3 & 3.4.

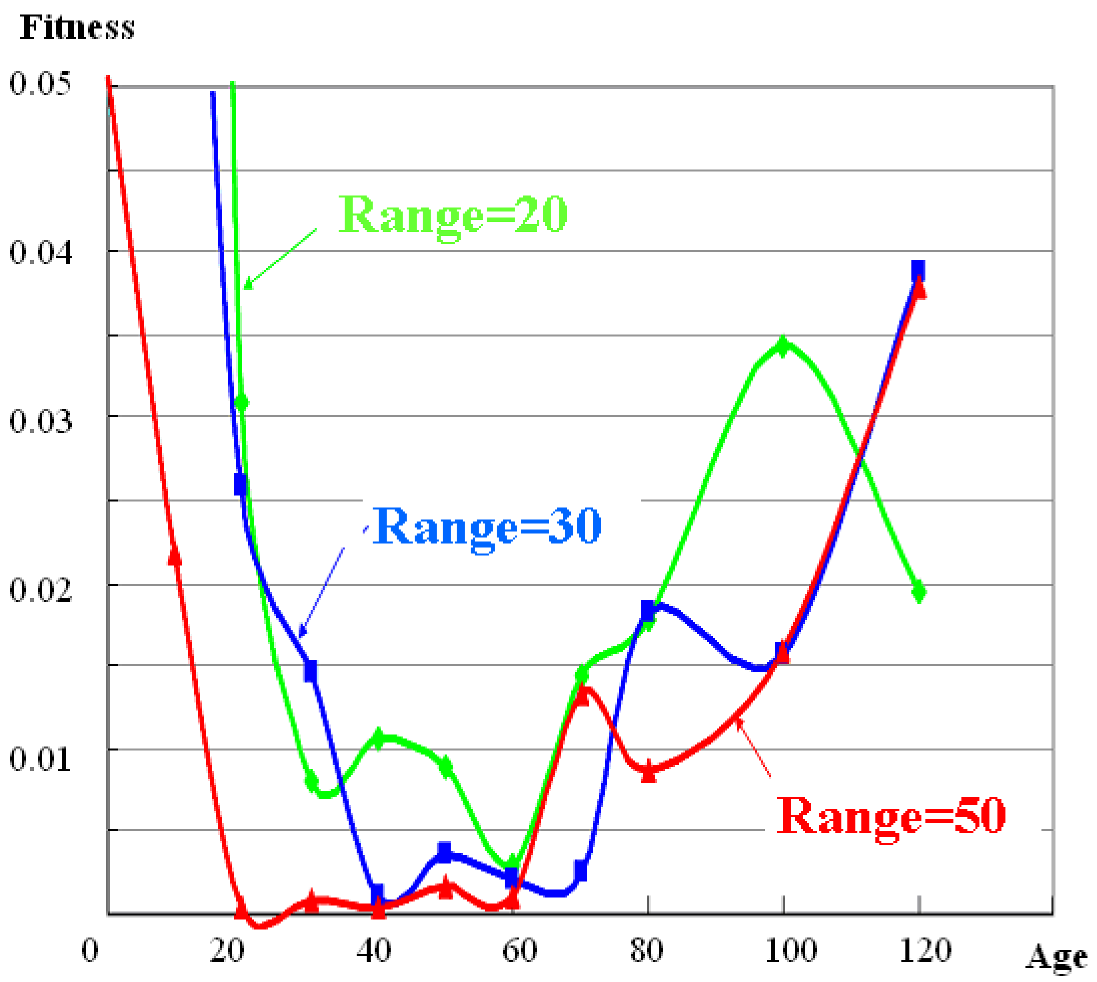


Figure 4.2: Average best fitness (20 runs) comparison among three different breeding age range and different starting breeding age.

Figure 4.2 shows the results of implementing different breeding ages. The x axis represents the ages at which fertility commences and the y axis stands for the average best fitness obtained from 20 runs (zero is the optimal function solution). The three lines represent three different ranges of fertility, these being 20, 30 and 50 time-steps. For example, the first data point from the left hand side on the red line at approximately Age =10, Fitness = 0.022) means, the average best fitness in 20 runs is 0.022, when the breeding starts from age 10 and the breeding range determined as 50 time-steps (age 10 to 59). .

As is shown by the results in Fig. 4.2, delaying the breeding age somewhat is beneficial but beyond roughly 60 time steps the benefits disappear. At 60 time steps, it is not too early, and not too late, just at half of the life span to avoid children paired with their parents. Also it is better to have a relatively lengthy period of fertility with the best results in fig.4.2 being obtained for 50 time-steps.

4.4 Family Tree - Champion Family and Mutation in Single CGA

As the aim to see how a best solution is created, a citizen number has been given to each individual. By tracking the best result’s family tree, there are two things which catch attention. One is mutation is which has not taken the most important role in the whole program. Another is that best result always comes from the champion family. The above results indicate that delayed breeding is beneficial, so it is instructive to examine how the breeding process works by providing a family tree for the “best” solution. This has been achieved by allocating a citizen number to each individual. The tracking of the best result’s family ancestors show two important factors. The first is that mutation is not a particularly significant factor although this could be a test problem specific feature. The second is that best result always comes from the champion family. These two factors will be explained in a minute.

To enable the family tree to be deciphered, it is necessary to explain the coding used. This is as follows:

$$\begin{tikzcd}[column sep=small] \text{(Age)} & \text{(Age)} & \text{------age to crossover} \\ \underline{\text{Parent 1 *}} \arrow[d, "\text{Time-step}"] & \underline{\text{Parent 2}} & \text{------parents citizen numbers} \\ & & \text{------time-step for crossover} \\ \text{Child} & & \text{------child citizen number} \end{tikzcd}$$

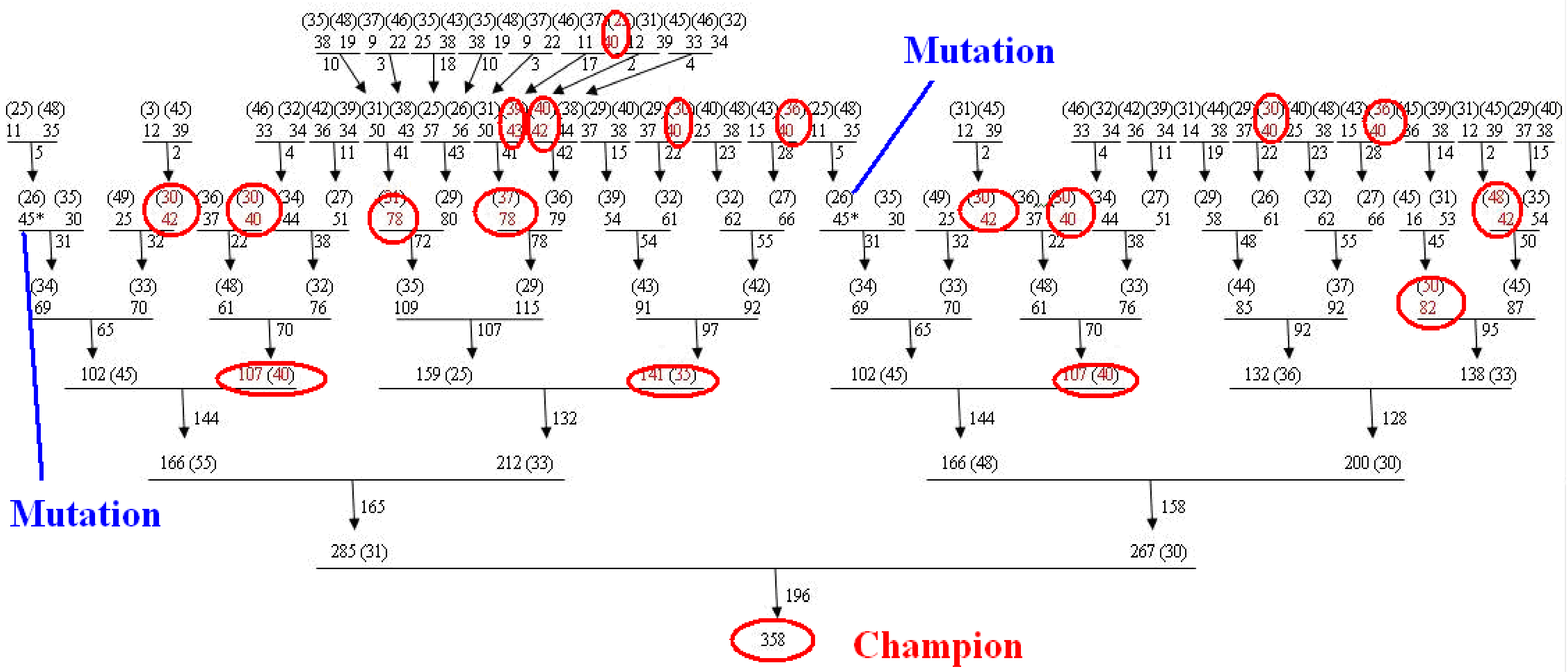


Figure 4.3: Family tree for first fit solution (De Jong's fifth test function)

Red circle implies this individual is used to be a Champion during the time steps

* means child is produced both through mutation and crossover, otherwise only crossover.

Fig. 4.3 shows the family tree for first fit solution, which shows how the first fit solution individual produces. All its former generations are listed in this family tree. The individuals on the top without arrows pointed to are from initial population. Not all of the initial population are taking part in the best solution creating process. But quite a few of them are repeatedly chosen as parents in this champion family.

Also from Fig. 4.3, it can be seen that many family members of the final solution were previous champions. In another words, the final result's family comes from a succession of previous fittest individuals for a given time step(s). And also it can be seen from Fig. 4.3 that some of the individuals, even not local champion, did crossover several times in different time-step, which shows one of the benefit of CGA – letting individuals remain in the population for quite a long time. Also the family tree, despite the relatively large amount of crossover, only two mutations happened. I have tried the higher mutation rate, but the results did not have much difference. It indicates that mutation is not a major feature for the given test problem.

4.5 Chapter Summary

In this chapter, we analysed factors which influence the single population CGA's performance, In CGA, population is very sensitive when choosing close population creation rate. Population creation rate controls the population size in order to get more steady search and children numbers. And effective-fitness type had chosen together with population creation rate range affect the speed of solution finding. In addition, mutation operator in CGA will help the solution finding. But compared with crossover, mutation is not a decisive action.

Chapter 5 Multiple Population CGA

5.1 Chapter Introduction

As mentioned in section 2.5, parallel genetic algorithms have been introduced to develop better GA performance to solve difficult problems. Erick (1995) gives a summary of research on Parallel Genetic Algorithms, borrowing an idea from which, I tried the multiple population in CGA (Chen, 2010). In this section, the performance of a two population CGA is examined. The motivation for trying a two population GA comes from the wish to model species' real life behaviour when competing for resources and to see what impacts this will have on the behaviour and performance of CGA. This is possible because the fundamentals of CGA are much closer to nature (e.g. the introduction of parameters such as 'age', effective fitness and life-span in chapter 3.3). Hence, the fundamental principle of multiple populations with CGA is not simply copying the framework of a "normal" multi-population GA. It is more like species competition in the real world.

The reason we named this new method 'two monkey species CGA' is because for each population, only the parameters modelling the monkey species were used in the current tests (Monkey modelling, see Chapter 3.4).

In this chapter, firstly there are several new features in the two-monkey CGA (TM-CGA), apart from the Single CGA, which need to introduce specifically. Secondly, compared with the single GGA, there is a brief explanation of the framework of TM-

CGA in section 5.3. Finally, the behaviours comparison between Single CGA and Two Monkey CGA is described at the end of this chapter.

5.2 New Parameters in TM-CGA

The novelty of the TM-CGA is that unlike the normal multiple populations GA, there is no combination or exchange between the two populations. The only shared aspect is they are both under the same pressure of there being a limited resource which we call “water”.

In addition to the features like age, life-span and etc. in single CGA, there are several things newly introduced in two-monkey CGA, such as “water”, “illness” and “accident”.

In each time-step, the population is decreased both through age-based deletion and accidental death. Age-based deletion is simply the removal from the population of an individual whose age exceeds its life-span. Besides, the second type of deletion is used whereby a small proportion of the population suffers “accidental death”. The risk of “accident” is according to the illness parameter of each individual, which is further related to the fitness.

Under the pressure of limitation resource - “water”, once the total population size of two populations is over “water” supply (i.e. in the current test, this is set at 120 individuals), all the individuals from 121 upwards, chosen on the basis of low fitness have their illness parameter incremented. This means only the 120 fittest individuals can avoid increasing their illnesses. The random accident selector generates a random number and then an individual is selected at random. If this individual's illness is greater than the random number then it is deleted from the population; so called death by "accident".

So for each time step of TM-CGA there are two pressures on the population size these being:

1. The population is increased through crossover and mutation; the population creation rate controls the speed of increase (chapter 4.2).
2. The population is decreased through the deletion of individuals, the latter being through age by life-span exceeding deletion and "accident" deletion.

5.3 Framework of TM-CGA

An understanding the parameters used in this new method makes it easier to understand the framework of TM-CGA. So the framework can be represented as follows:

1. Two populations are acting as single CGA completely separately.
2. When the total number of individuals of the two populations exceeds the “water” supply, for the numbers of the weaker individuals (i.e. the least fit) who exceed the limit population allowed for the water supply are given an increased rate of illness.
3. Randomly selected individuals are subjected to an accident, the risk of which is according to their illness parameter, i.e. an individual with a high illness parameter is more likely to be killed off.

After the accident deletion, go back to step 1.

To make it more vivid and accessible, Fig 5.1 shows the framework in a different way. As shown in Fig 5.1, the process starts with the creation of the initial population. As in the single CGA (chapter 3.2), each individual is given an age, fitness, effective fitness and a life span, but one more parameter will be given as well – illness rate, which is set at zero in the beginning. The two sub-CGAs are using exactly the same initial population. The process then starts to work around the main loops, but in two separate sub-populations. The thick circle line indicates the majority of the population, and the thinner line shows deletion or crossover, etc. operators mean only a small number of the individuals are taking part.

All the main processes in the sub-CGA are almost the same as that in single CGA (Fig 3.1). Things needed to pay attention to are:
The box which connects the two circles together is the key point of TM-CGA and is the only point of intersection between two sub-populations. "Water" pressure will be added here to all the two populations. Illness rate for the lower fitness individuals will be increased when the total population exceeds the "water" supply.

Accident deletion is the other way to reduce the population size apart from life-span exceeding deletion. A random number will be chosen for accident deletion. Every individual will be comparing its illness rate with that accident number. If the illness rate is bigger than that number, this individual will be deleted from the population. So when the total population goes beyond the "water" limit, it will result in increasing the illness rate, which makes the poorer individual have a higher risk on the accident deletion scale.

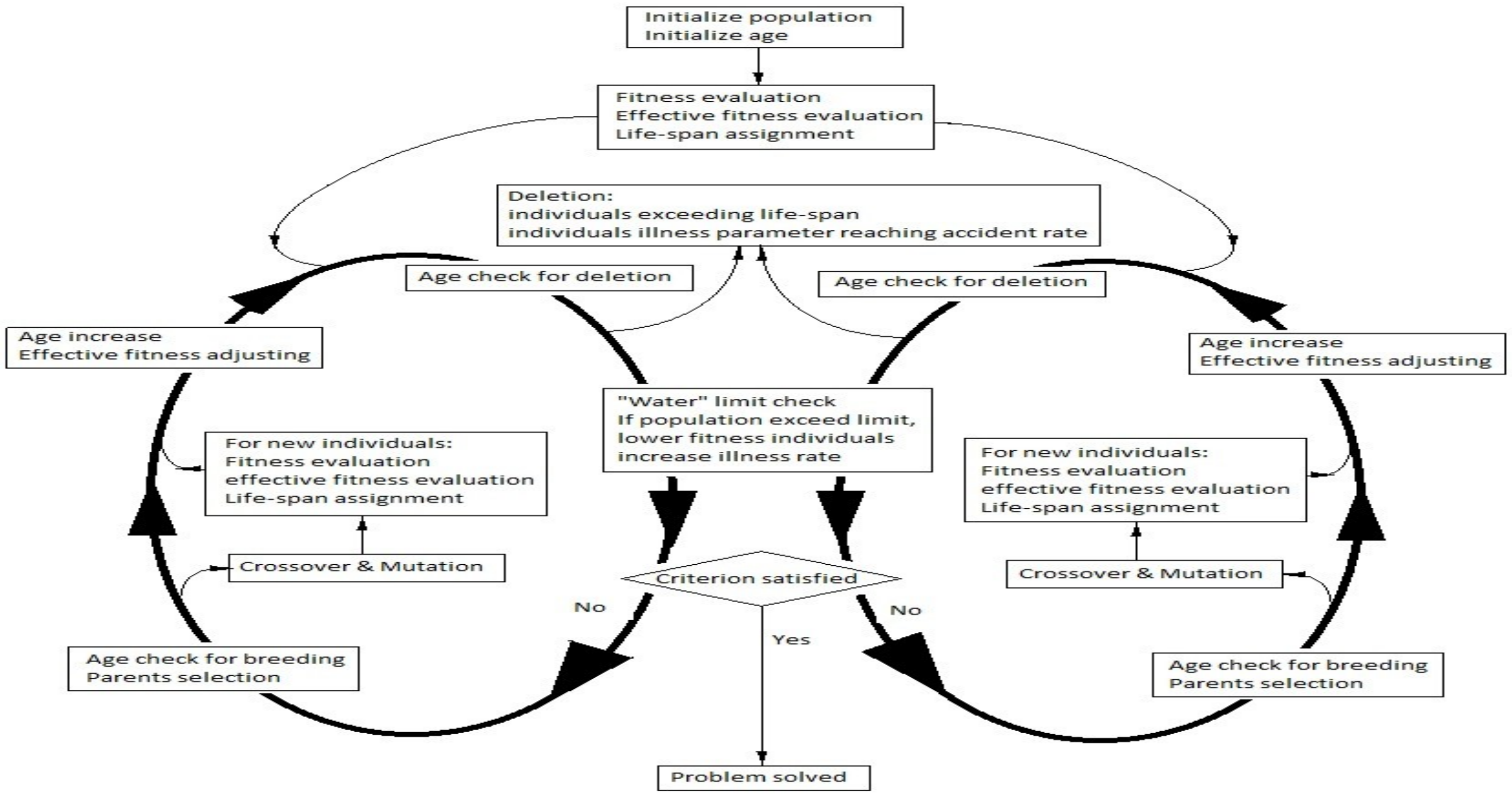


Figure 5.1: Framework of two populations CGA

5.4 Behaviour of TM-CGA

Accepting the new parameters and the framework of TM-CGA, the behaviours of this two population CGA will be shown and discussed in this section. And also the comparison between single CGA and TM-CGA will be listed in the later thesis.

De Jong's fifth test function (details in chapter 3.5.2.2, Table 3.6, 3.7 and 3.5) is used in testing. Ordinary binary coding and tournament selection criterion have been used. The initial size of each population is 40 and life-span values have been assigned according to the fitness ranking in each population (60 was the maximal life-span for the fittest individual, while 40 was the minimal life-span for the worst one, and all the others were in between according to their fitness by linear scaling). Every individual was also given a random age in the range zero to 50 time-steps. The population creation rate was 4% of population.

5.4.1 Performance of TM-CGA

Figs. 5.2 & 5.3 below illustrate the behaviour of the two-Monkey CGA for an optimization using a two-variable De Jong's function.

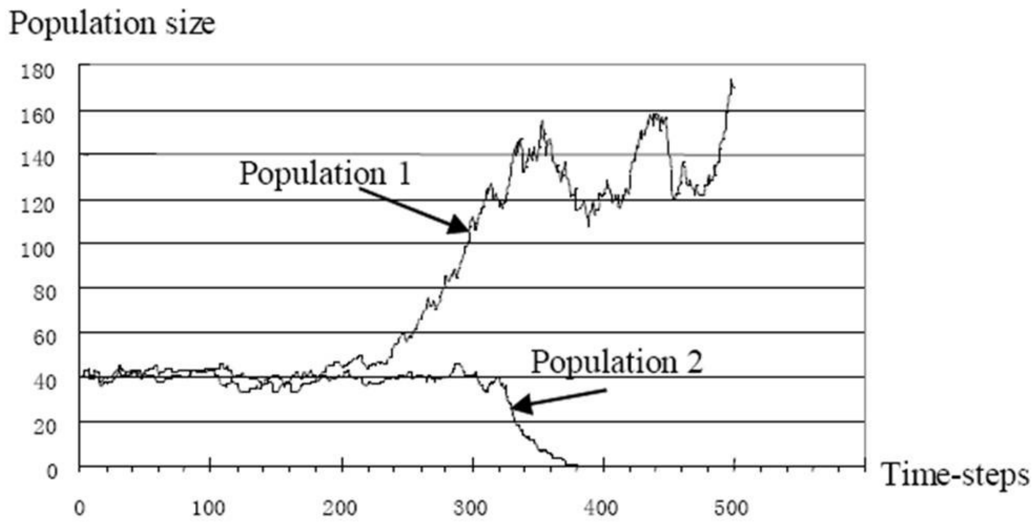


Figure 5.2: Population size changes with time-steps in TM-CGA (De Jong's function).

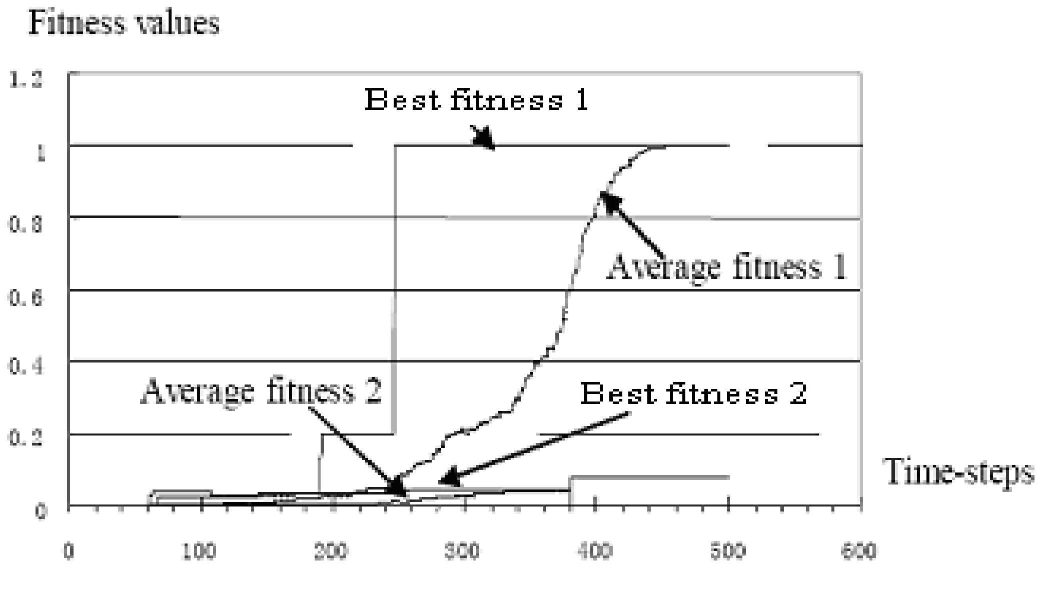


Figure 5.3: Best fitness individual from population 1 & 2 and average fitness values of two sub-populations along with time-steps (De Jong's function).

Fig. 5.2 shows the changes in population size for both population 1 and 2 during 500 time-steps. The two populations were identical in the beginning. As can be see from around time-step 200, the size of population 1 started to increase. After checking with the individual output record and the fitness graph (Figure 5.3), there was a hypothesis that the reason for this change is because population 1 generated one or more highly fit individuals and this resulted in an overall increase in the fitness of the population. The way that the accident rate works is that the life-span of fitter individuals tends to be somewhat longer than the less fit which means they have an enhanced chance of breeding and therefore increasing the overall fitness of the population. It can be seen from Fig. 5.3 that the average fitness of population 1 does start to become significantly better than that of population2 from around time-step 280, when the total population size is over 120 (population 1 is around 80, population 2 about 40). As this exceeds the "water" limit, the weaker individuals suffer increasing levels of illness which lead to a high accident risk. This causes the rapid drop in the size of population 2 after 320 time-steps.

Fig. 5.3 shows the best and average fitnesses of population 1 and 2 over the 500 time-steps. The best fitness of population 1 shows a significant increase at about 250 time-step when an individual representing the optimum answer is produced. Also from tracking all the individuals, the family tree of the first best individual can be made. The results show that the best solution came mostly from the champions of previous time-steps. Almost all the local champions appeared in the best result's family. In addition, there are not many of the family members which have been produced through mutation (only one member has been mutated). This indicates that for this problem and form of GA, the best results are produced mostly from crossover. And this is not a special case, most of the runs show the same characteristic.

5.4.2 Competition between Single CGA and TM-CGA

The comparison of performances between Single CGA and Two-Monkey CGA is shown in Table 5.1. The performance of TM-CGA has been compared to that of an equivalent Single CGA for De Jong's function. The function has two variables, and it was encoded with 34 bit binary strings to adequately cover a search space of -65.535 to 65.535 with three decimal places. Both the S-CGA and TM-CGA have been programmed to run on the same computer using C++, so that the performance of the two GAs can be directly compared.

In Table 5.1, comparisons for these two methods are based on: failure rate and the number of evaluations that have been produced to enable the program to locate a satisfactory result. Failure rate means in 15 times random runs, how many unsuccessfual runs can occur within 500 time-steps. Number of evaluations means when an acceptable result appeared, such number of children have been created, this is one way to consider how many genetic operations have occurred. The results of the comparison are quite interesting. Althought the successful rate of S-CGA (0.04) is smaller than the TM-CGA (0.8), a study of the number of evaluations shows TM-

CGA, when successful, locates the answer with less effort (i.e. fewer children) than the S-CGA.

Table 5.1: Performance of Single CGA and Two-Monkey CGA for De Jong's test function.

Failure rate	Single CGA			
	Number of evaluations of the success runs			Average number of evaluations
0.04	797	872		754
	762	951		
	679	570		
	714	560		
	773			
0.8	Two-Monkey CGA			
	Population 1	Population 2	Total	Average
	567	**329 (best)**	896	654
	272 (best)	263	535	
	270 (best)	261	531	

5.5 Chapter Summary

The newly proposed TM-CGA has been described in this Chapter. It shows an improvement on the single CGA in some respects because it offers a new sort of framework for multiple populations GA which is much closer to modelling the real world. Tests in this thesis have shown that although TM-CGA has a higher failure rate than the S-CGA, but it requires fewer genetic operations and is overall computationally faster than S-CGA when it does locate the correct answer.

Chapter 6 Discussion and Conclusion

In this thesis, after reviewing the history and development of GAs, we got a brief idea of how ordinary GA works. Chapter 3, 4, and 5 gave more information of the GA method – Cardiff Gigantic Algorithm (CGA). Brief ideas of single population CGA were described. This is a genetic algorithm that mimics natural population. Age, life span, population creation rate, etc., are many of the familiar names that have been used in the CGA modelling program. By using those parameters, a new framework of CGA was built. By giving different values to the new parameters, a variety of species can be mimicked. And the result shows a longer life span and lower population creation rate just like monkey model has a better performance in some specific functions. From the results, this newly proposed CGA has been shown an improvement by giving more diversity within the searching space. It offers a new sort of framework for GAs which is much closer to the real world.

Later on, we analyse factors which influence the single population CGA's performance. In CGA, population is very sensitive when choosing close population creation rate. Population creation rate controls the population size in order to get a more steady search and children numbers. Effective-fitness type chosen together with the population creation rate range affects the speed of solution finding. In addition, the mutation operator in CGA will help the solution finding. But compared with crossover, mutation is not a decisive action.

The newly proposed two-population CGA, Two-Monkey CGA (TM-CGA) is shown last. It shows an improvement on the single CGA in some respects because it offers a new sort of framework for multiple populations GAs which is much closer to modelling the real world. Tests in this thesis show that TM-CGA has a higher failure rate than the S-CGA. But it requires fewer genetic operations and is overall computationally faster than S-CGA when it does locate the correct answer. This is interesting because although there is only a very loose coupling between the two populations with no exchange of genetic material, the presence of two populations influences the search. The main influence is through the illness parameter.

It is a new approach for this fundamentally different method. Although improvement in some test functions were seen, there is still much more to be explored and tested in order to see the full picture of CGA. Further implementations and testing are necessary. But the proposed CGA-based algorithms offer an interesting and novel alternative to other forms of GA.

Future work in Civil Engineering

In civil engineering, genetic algorithms are helpful in many areas during this decade, such as optimization of structure design (Jenkins, 1991), detecting structural damage (Au, 2003), pipe network optimization (Zheng, 2013) and so on. CGA, as a new family member of GAs, can be experimented in these areas. However, because different problems have different requests and situations, there is no single method which solves all of the problems. It is still good news that there is a different approach available and hope that this CGA will fit some of them.

References

[1]	Ackley, D.H., 1987. A connectionist machine for genetic hill climbing. Kluwer Academic Publishers, Boston.

[2]	Balling, R. 2003. The Maximin Fitness Function; Multi-objective City and Regional Planning. Fonseca, C.M., Fleming, P. J., Zitzler, E., Deb, K. and Thiele, L (eds). *Evolutionary Multi-criterion Optimization: Second International Conference, EMO 2003, Faro, Portugal, April 8-11, 2003: Proceedings.* Publisher: Springer.

[3]	Buckles, B. P. and Petry, F. E. 1992. Genetic Algorithms. Los Alamitos: IEEE Computer Society Press.

[4]	Chen, H., Miles, J.C. and Kwan, A.S.K. 2008. CardiffGA: A new Genetic Algorithm framework, Proceedings of *6th Int. Conf. on Engineering Computational Technology* (Eds: M. Papadrakakis and B.H.V. Topping), Civil-Comp Press, p77.

[5]	Chiras, D. D. 2006. Environmental Science. Seventh edition. America: Jones & Bartlett Publishers.

[6]	Coello, C.C.A, 2006. Evolutionary multi-objective optimization: a historical view of the field. *Computational Intelligence Ma*gazine, IEEE, Volume: 1 , Issue 1. Ieeexplore.ieee.org

[7]	Davis, L. and Steenstrup, M. 1988. Genetic Algorithms and Simulated Annealing: An Overview. In Davis, L. (Ed) *Genetic Algorithms and Simulated Annealing*. London: PITMAN PUBLISHING.

[8]	Deb, K. 1999. Multi-objective Genetic Algorithms: Problem Difficulties and Construction of Test Problems in *Evolutionary Computation*, volume 7, pp 205-230, Massachusetts Institute of Technology.

[9]	Deb, K., Pratap, A, Agarwal, S. and Meyarvan, T. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II in IEEE Transactions *on Evolutionary Computation*, volume 6, issue 2, pp 182-197. IEEE.

[10]	De Jong, K. 1975. An analysis of the behaviour of a class of genetic adaptive systems. PhD thesis, University of Michigan.

[11] Dorigo, M., Maniezzo, V. and Colorni, A., 1996. Ant System: Optimization by a Colony of Cooperating Agents, IEEE Transactions on *Systems, Man, and Cybernetics*–Part B, 26 (1): 29–41.

[12] Erick Cant´u-Paz. 1997. Designing efficient master-slave parallel genetic algorithms. IllGAL Report 97004, The University of Illinois, Available on-line at: ftp://ftp-illigal.ge.uiuc.edu /pub/ papers/IlliGALs/97004.ps.Z.

[13] Erick Cant´u-Paz. 1998. Designing scalable multi-population parallel genetic algorithms. IllGAL Report 98009, The University of Illinois, Available on-line at: ftp://ftp-illigal.ge.uiuc.edu/pub/ papers/IlliGALs/98009.ps.Z.

[14] Eshelman, L. J. 1991. The CHC Adaptive Search Algorithms: How to Safe Search When Engaging in Nontraditional Genetic Recombination. Rawlins, G. J. E. (ed) *Foundations of Genetic Algorithms*. USA: Morgan Kaufmann Publishers, Inc.

[15] Falkenauer, E. 1998. Genetic Algorithms and Grouping Problems. West Sussex: John Wiley & Sons Ltd.

[16] Fogarty, T. C. and Huang, R., 1991. Implementing the genetic algorithm on transputer based parallel processing systems, *in Parallel Problem Solving from Nature*, Berlin, Germany, 1991, pp. 145–149, Springer Verlag.

[17] Goldberg, D. E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Canada: Addison Wesley Longman, Inc.

[18] Goldberg, David E, 1989. Genetic Algorithms in Search, Optimization and Machine Learning, Kluwer Academic Publishers, Boston, MA.

[19] Grefenstette, J. J., De Jong, K. A., and Spears, W. M. 1993. Competition-Based Learning. Chipman, S. F. Meyrowitz, A. L. (eds) *Foundations of Knowledge Acquisition: Machine Learning*. Publisher: Springer.

[20] Haupt, R. L., Haupt, S. E. 2004. Practical Genetic Algorithms. Second edition. USA: Wiley-IEEE.

[21] Holland, John H, 1975. Adaptation in Natural and Artificial Systems. America University of Michigan Press, Ann Arbor.

[22] Horton, I. 1998. Beginning VisualC++ 6, Birmingham: Wrox Press.

[23] Hu, J. and Fu, M. and Marcus, S. 2008. A Model Reference Adaptive Search Method for Global Optimization. Available at: http://reference.kfupm.edu.sa/content/a/d/adaptive_global_optimization_with_local_ _433291.pdf.

[24] Kehoe, A. B. 1998. Humans: An Introduction to Four-field Anthropology. London: Routledge.

[25] Keeney, R. Raiffa, H. and Rajala, D. 1976. Decisions with Multiple Objectives: Preferences and Value Trade-offs in *IEEE Transactions on Systems, Man and Cybernetics*, volume 9, issue 7, p403, IEEE.

[26] Kennedy, J.; Eberhart, R., 1995.Particle Swarm Optimization". Proceedings of *IEEE International Conference on Neural Networks*. IV. pp. 1942–1948. http://www.engr.iupui.edu/~shi/Coference/psopap4.html.

[27] Ladd, S. R. 1996. Genetic Algorithms in C++. New York: M&T Books.

[28] Machwe, A. and Parmee, I., 2007. Multi-objective analysis of a component based representation within an interactive evolutionary design system.*Engineering Optimization*, 39 (5). pp. 591-615. ISSN 0305-215X.

[29] Meyers, S. 2005. Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition). USA: Addison-Wesley Professional.

[30] Michalewicz, Z. 1992. Genetic Algorithms + Data Structures = Evolution Programs. United States of America: Springer-Verlag Berlin Heidelberg.

[31] Michalewicz, Z. 1996. Genetic Algorithms + Data Structures: Evolution Programs. Third edition. Publisher: Springer.

[32] Mitchell, M. 1998. An introduction to Genetic Algorithms. Cambridge: The MIT Press.

[33] Perry, G. M. 2001. Sams Teach yourself beginning programming in 24 hours. Second edition. Australia: Sams Publishing.

[34] Pham, D. T., Ghanbarzadeh A., Koç E., and Otri, S., 2006. Application of the Bees Algorithm to the training of radial basis function networks for control chart pattern recognition, Proc*5th CIRP International Seminar on Intelligent Computation in Manufacturing Engineering* (CIRP ICME '06), Ischia, Italy.

[35] Raphael. B. and Smith. Ian F. C. 2003. Fundamentals of Computer-Aided Engineering. West Sussex: John Wiley & Sons Ltd.

[36] Reeves, C. R. 1994. Genetic Algorithms and neighbourhood Search. Fogarty, T. C. (ed) *Evolutionary Computing: AISB Workshop, Leeds, U.K., April 11-13, 1994 : Selected Papers*. UK: Springer.

[37] Schildt, H. 1994. *C++ from the Ground Up*. Berkeley, California: Osborn McGraw-Hill.

[38] Schildt, H. 2008. *C# 3.0: A Beginner's Guide*. Second edition. New York: McGraw-Hill Professional.

[39] Schraudolph, N. and Belew, R. 1992. "Dynamic parameter encoding for genetic algorithms" in *Machine Learning,* volume 9, pp. 9-21, Springer Netherlands.

Spears, W. M. and De Jong, K. A, etc. 1993. An overview of evolutionary computation. In *Computer scienc*e, Volume 667/1993.

[40] Wolpeart, D and MacReady, W., 1997. No free lunch theorems for optimization, IEEE Trans. On *Evolutionary Comp 1(1)*. p67-82. http://ic.arc.nasa.gov/people/dhw/papers/78.pdf

[41] Yao, X. and Liu, Y. 1998. Making use of population information in evolutionary artificial neural networks in *IEEE transactions onSystems, Man, and Cybernetics, Part B: Cybernetics*, volume 28, issue 3, pp 417-425, IEEE.

[42] Zalzala, A. M. S. and Fleming, P. J. 1997. *Genetic Algorithms in Engineering* Systems. UK: The institution of Electrical Engineers.

Appendix

Code for the programming

A. Initialize program (Ackley)

```
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <cmath>
#include <string>
#include <iomanip.h>
#include <fstream>

using namespace std;

void sort();
double bin2dec(int,int);
double bin2dec2(int);
double fit(int);

int individual[500][54];
int age[500];
int swap1;
int swap2;
int swap3;

ofstream fout("out.txt");

main()
{    fout << "The initial numbers are: \n";
     int c1,c2,d;
     int i,j;
     srand((unsigned)time(NULL));

     for(i=0; i<120; i++)
     {         for(j=0; j<15; j++)
               {         c1=rand()%2;
                         c2=rand()%2;
                         individual[i][j]=c1;
                         individual[i][j+26]=c2;
                         d=rand()%200;
                         age[i]=d;
               }}

     sort();

     for(i=80; i<120; i++)
     {         for(int j=0; j<54; j++)
```

```
            individual[i][j]=0;
            age[i]=0;
    }

    for(i=0; i<80; i++)
    {       fout<<bin2dec(1,i)<<" "<<bin2dec(2,i)  <<"{"<<fit(i)<<")"<<"   ";
    }

    fout << "\n\n";

    fout<<"initial[80][54]={";
    for(i=0; i<80; i++)
    {       fout<<",{";
            for(j=0; j<54; j++)
            {       fout<<individual[i][j]<<",";}
            fout<<"}\n                ";
    }
    fout<<"}\n";

    fout<<"\ninitial_age[80]={";
    for(i=0; i<80; i++)
    {       fout<<age[i]<<",";}

    return 0;
}
```

```
void sort()
{    for(int i=0; i<120; i++)
     {         for(int j=i+1; j<120; j++)
               {         if(fit(j)>fit(i))
                         {for(int m=0; m<54; m++)
                              {         swap2=individual[i][m];
                                        individual[i][m]=individual[j][m];
                                        individual[j][m]=swap2;
                              }

                              swap3=age[i];
                              age[i]=age[j];
                              age[j]=swap3;
                         }}}}

double fit(int i)
{    double fitness=0;
     double x1=(double)bin2dec(1,i);
     double x2=(double)bin2dec(2,i);

     fitness=(double)exp(-
0.2)*(double)sqrt((double)pow((double)x1,2)+(double)pow((double)x2,2))+(double)3*(
(double)cos((double)2*(double)x1)+sin((double)2*(double)x2));
     return fitness;
}

double bin2dec(int m,int i)
{double dec=0;
    int j;
    if(m==1)
    {for(j=1; j<27; j++) (double)dec+=(double)(individual[i][j])*(double)pow(2.0,(26-
j));

            if(individual[i][0]==0)   return (double)dec/10000000;
            else      return -(double)dec/10000000;
    }
    else if(m==2)
    {for(j=1; j<27; j++)
(double)dec+=(double)(individual[i][j+27])*(double)pow(2.0,(26-j));

            if(individual[i][26]==0)  return (double)dec/10000000;
            else      return -(double)dec/10000000;
    }
    else      {for(j=0; j<54; j++)
                    (double)dec+=(double)(individual[i][j])*(double)pow(2.0,(53-j));
            return (double)dec/1000;
    }}
```

B. Single CGA (Ackley)

```
#include <stdlib.h>

#include <iostream.h>

#include <stdio.h>

#include <time.h>

#include <math.h>

#include <cmath>

#include <string>

#include <iomanip.h>

#include <fstream>

#include<time.h>


using namespace std;


void initialize();

void generation();

void deletion();

void accident();

void putout(int);

void printout(int);

double child_number();

void select_parent(int);

void crossover(int);

void new_generation(int);
void mutation(int);

void new_individual();

void average(int);

double bin2dec(int,int);

double bin2dec2(int);

double fit(int);

double effective_fitness(int);

double val1(double);

double val2(double);

void alivenumber(int);

void sort(int);

void re_initial();

void migration();


int g;

int initial[80][54]={};        //get from initialize program

int individual[500][54];

int initial_age[80]={};        //get from initialize program

double fitness[1000];

int sort_number[1000];

double illness[1000];

int age[500];

int life_span[500];

int swap1;

int swap2;
int swap3;
int swap4;

double best_x1;
double best_x2;
int best_illness;
int best_age;
double best_fitness;
double total_fitness;
double average_fitness;
double parent_average;
double parent_best1;

int new_gene[500][54];
int parent_number[500];
int parent[500][54];
int parent_n[500];
int parentnumber=0;
int Chld[500][54];
int age_new_gene[500];
int new_life_span[500];
double new_illness[1000];
double f;
int accident_number;
int over_lifespan;
int mutate_gene;
int alive_number;

double range;

int number;
ofstream tout("table.txt");
main()
{    srand((unsigned)time(NULL));
     initialize();
     g=0;
     printout(g);
     average(g);
     for(int i=0; i<80; i++)
     {          if(fit(i)>average_fitness)
                {          life_span[i]=195-int((fit(i)-average_fitness)/range);
                           if(life_span[i]<190)
                                      life_span[i]=190;
                }
                else
                {          life_span[i]=195+int((average_fitness-fit(i))/range);
                           if(life_span[i]>200)
                                      life_span[i]=200;
                }}
```

```
    putout(g);

    do
    {g++;
    generation();
            deletion();
            accident();
            average(g);
            printout(g);
            putout(g);
    }while(g!=500);
    return 0;
}

void initialize()
{   int i,j;
    for(i=0; i<80; i++)
    {       for(j=0; j<54; j++)
            {individual[i][j]=initial[i][j];}
            age[i]=initial_age[i];
            illness[i]=1;
    }}

void alivenumber(int g)
{   for(int i=0; bin2dec(3,i)!=0 && i<500; i++)
```

```
    {alive_number=i+1;}
}


void average(int g)
{    total_fitness=0;
     average_fitness=0;
     best_fitness=fit(0);
     best_x1=bin2dec(1,0);
     best_x2=bin2dec(2,0);
     best_illness=illness[0];
     best_age=age[0];
     number=0;

     alivenumber(g);

     for(int i=0; i<alive_number; i++)
     {          total_fitness+=(double)fit(i);
                if(fit(i)<best_fitness)
                {          best_fitness=fit(i);
                           best_x1=bin2dec(1,i);
                           best_x2=bin2dec(2,i);
                           best_illness=illness[i];
                           best_age=age[i];
                           number=i;
                }}
     average_fitness=(double)total_fitness/(double)i;

     range=(average_fitness-best_fitness)/5;

}




void putout(int g)

{   tout<< g <<" "<<alive_number<<" alive, ";

     tout<<f<<" Children, ";

     tout<<accident_number<<" accident, ";

     tout<<over_lifespan<<" lifespan, ";

     tout<<"average_fitness: "<<(double)average_fitness;

     tout<<", best_fitness: "<<best_fitness;

     tout<<" ("<<best_x1<<", "<<best_x2<<", "<<best_age<<") ";

     tout<<"\n";

}


int test_function()

{   if(best_fitness<(-0.417946) )

                return 1;

     else

                return 0;

}


double fit(int i)
```

```
{    double fitness1=0;
     double x1=(double)bin2dec(1,i);
     double x2=(double)bin2dec(2,i);

     fitness1=(double)exp(-
0.2)*(double)sqrt((double)pow((double)x1,2)+(double)pow((double)x2,2))+(double)3*(
(double)cos((double)2*(double)x1)+sin((double)2*(double)x2));
     return fitness1;
}

double val1(double x1,double x2)
{    double value1=0;
     value1=(double)x1*(double)exp(-
0.2)/(double)sqrt((double)pow((double)x1,2)+(double)pow((double)x2,2))-
(double)6*sin((double)2*(double)x1);
     return value1;
}

double val2(double x1,double x2)
{    double value2=0;
     value2=(double)x2*(double)exp(-
0.2)/(double)sqrt((double)pow((double)x1,2)+(double)pow((double)x2,2))+(double)6*c
os((double)2*(double)x2);
     return value2;
}
```

```
double effective_fitness(int i)
{    double c;
     if(age[i]>50 && age[i]<=150)    c=(double)(1.00);
     else      c=(double)(0);
     return c;
}
double bin2dec(int m,int i)
{double dec=0;
     int j;
     if(m==1)
     {          for(j=1; j<27; j++)
                     (double)dec+=(double)(individual[i][j])*(double)pow(2.0,(26-j));
                if(individual[i][0]==0)   return (double)dec/10000000;
                else      return -(double)dec/10000000;
     }
     else if(m==2)
     {for(j=1; j<27; j++)
                (double)dec+=(double)(individual[i][j+27])*(double)pow(2.0,(26-j));
                if(individual[i][26]==0)  return (double)dec/10000000;
                else      return -(double)dec/10000000;
     }
     else
     {for(j=0; j<54; j++)        (double)dec+=(double)(individual[i][j]);
                return (double)dec;
     }}
```

```
double bin2dec2(int i)
{    double dec=0;
     int j;
     for(j=0; j<54; j++)          (double)dec+=(double)(new_gene[i][j]);
     return (double)dec;
}

void printout(int g)
{    cout<<g<<": \n";}
void generation()
{    re_initial();
     f=child_number();
     if(f==0)
     {          new_generation(0);
                new_individual();
     }
     else if(f==(-1))
     {          for(int i=0; i<500; i++)
                          for(int j=0; j<54; j++)
                                    individual[i][j]=0;
     }
     else
     {          select_parent(f);
                crossover(f);
```

```
            new_generation(f);
            new_individual();
    }}

void re_initial()
{   parentnumber=0;
    parent_average=0;
    parent_best1=0;
    for(int i=0; i<500; i++)
    {       parent_number[i]=0;
            parent_n[i]=0;
            mutate_gene=0;
            age_new_gene[i]=0;
            new_life_span[i]=0;
            new_illness[i]=0;
            for(int j=0; j<54; j++)
            {       Chld[i][j]=0;
                    new_gene[i][j]=0;
            }}}


double child_number()
{   int f;
    for(int c=0; c<alive_number; c++)
    {       if(effective_fitness(c)>0)
```

```
        {       parent_number[parentnumber]=c;
                parentnumber++;
                parent_average+=fit(c);
                if(fit(c)<parent_best1)   parent_best1=fit(c);
        }}

    parent_average=parent_average/parentnumber;


    if(alive_number==0)     f=(-1);
    else if(parentnumber==0)    f=0;
    else if(alive_number>=160)  f=(int)((double)0.01*(double)alive_number);
    else      f=(int)((double)0.03*(double)alive_number);
    return f;
}

void select_parent(int f)
{   int m,p;
    int n,q;
    for(int i=0; i<f; i++)
    {       m=rand()%parentnumber;
            n=rand()%parentnumber;
            p=rand()%parentnumber;
            q=rand()%parentnumber;

            if(fit(parent_number[m])<fit(parent_number[n]))
        {for(int j=0; j<54; j++)
            {parent[2*i][j]=individual[parent_number[m]][j];}
        }
        else
        {   for(int j=0; j<54; j++)
            {parent[2*i][j]=individual[parent_number[n]][j];}
        }
        if(fit(parent_number[p])<fit(parent_number[q]))
        {for(int j=0; j<54; j++)
            {parent[2*i+1][j]=individual[parent_number[p]][j];}
        }
        else
        {for(int j=0; j<54; j++)
            {parent[2*i+1][j]=individual[parent_number[q]][j];}
        }}}
void crossover(int f)
{   int b;
    int i,j;
    for(i=0; i<f; i++)
    {       srand((unsigned)time(NULL));
            b=rand()%54;
            for(j=0; j<b; j++)
            {       individual[499][j]=parent[2*i+1][j];
                    individual[498][j]=parent[2*i][j];
            }
            for(j=b; j<54; j++)
            {       individual[499][j]=parent[2*i][j];
                    individual[498][j]=parent[2*i+1][j];
            }
            if(fit(499)<fit(498))
            {for(j=0; j<54; j++)      Chld[i][j]=individual[499][j];    }
            else      {for(j=0; j<54; j++)      Chld[i][j]=individual[498][j];}


            for(j=0;j<54;j++)
            {       individual[499][j]=0;
                    individual[498][j]=0;
            }}}
void new_generation(int f)
{   int i,j;
    for(i=0; i<alive_number; i++)
    {       for(j=0; j<54; j++)
            {new_gene[i][j]=individual[i][j];}
            age_new_gene[i]=age[i]+1;
            new_life_span[i]=life_span[i];
            new_illness[i]=illness[i];
    }
    for(int m=0; m<f; m++)
    {       for(j=0; j<54; j++)
            {new_gene[i][j]=Chld[m][j];}
            age_new_gene[i]=1;
            new_life_span[i]=500;

            new_illness[i]=1;

            mutation(i);

            i++;

    }}
void mutation(int i)
{   int b;
    srand((unsigned)time(NULL));
    if((rand()%1000)<50)
    {       b=rand()%54;
            new_gene[i][b]=abs(new_gene[i][b]-1);
    }}

void new_individual()
{   int i, j;
    for(i=0;i<500;i++)
    {       for(j=0;j<54;j++)       individual[i][j]=0;
            age[i]=0;
            life_span[i]=0;
            illness[i]=1;
    }
    for(i=0; i<500; i++)
    {       for(j=0; j<54; j++)
            {individual[i][j]=new_gene[i][j];}
            age[i]=age_new_gene[i];
```

```
                life_span[i]=new_life_span[i];

                illness[i]=new_illness[i];

        }}


void deletion()

{       int i;

        for(i=0; i<500; i++)

        {       if(life_span[i]==500)

                {if(fit(i)>average_fitness)

                        {       life_span[i]=195-int((fit(i)-average_fitness)/range);

                                if(life_span[i]<190)    life_span[i]=190;

                        }

                        else

                        {       life_span[i]=195+int((average_fitness-fit(i))/range);

                                if(life_span[i]>200)

                                life_span[i]=200;

                        }}}}


void accident()

{       int i,j,m,n,b;

        alivenumber(g);

        over_lifespan=0;

        for(i=1,m=0; m<alive_number; m++,i++)

        {       if(age[i-1]>life_span[i-1])

                {       for(n=i-1; n<=alive_number; n++)
                        {for(j=0; j<54; j++)  individual[n][j]=individual[n+1][j];
                        age[n]=age[n+1];
                        life_span[n]=life_span[n+1];
                        illness[n]=illness[n+1];
                        }
                        i--;
                        over_lifespan++;
                }}

        alivenumber(g);
        accident_number=0;

        for(i=1,m=0; m<alive_number; m++,i++)
        {       b = rand()%1000;
                if(b<10)
                {       for(n=i-1; n<=alive_number; n++)
                        {       for(j=0; j<54; j++)
                                        individual[n][j]=individual[n+1][j];
                                age[n]=age[n+1];
                                life_span[n]=life_span[n+1];
                                illness[n]=illness[n+1];
                        }
                        i--;
                        accident_number++;
                }}}
```

C. TM-CGA (De Jong)

```
#include <stdlib.h>
#include <iostream.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <cmath>
#include <string>
#include <iomanip.h>
#include <fstream>
#include<time.h>

using namespace std;

void initialize();
void change1();
void change2();
void change_back1();
void change_back2();
void generation1();
void generation2();
void deletion();
void accident();

void putout();

void printout(int);

double child_number1();

double child_number2();

void select_parent1(int);

void select_parent2(int);

void crossover1(int);

void crossover2(int);

void new_generation1(int);

void new_generation2(int);

void mutation1(int);

void mutation2(int);

void new_individual1();

void new_individual2();

void average1(int);

void average2(int);

double bin2dec1(int,int);

double bin2dec12(int);

double fit1(int);

double effective_fitness1(int);

double bin2dec2(int,int);

double bin2dec22(int);

double fit2(int);

double effective_fitness2(int);
double val1(double,double);

double val2(double,double);

void alivenumber1(int);

void alivenumber2(int);

void sort(int);

void re_initial1();

void re_initial2();


int g;

int initial1[40][34]={}; // get from initialize program

int initial2[40][34]={}; // get from initialize program


int initial_age1[40]={}; // get from initialize program

int initial_age2[40]={}; // get from initialize program

int individual1[500][34];

int individual2[500][34];

double fit[500];

int sort_number[500];

int citizen_number1[10000];

double illness1[1000];

int citizen_number2[10000];

double illness2[1000];

int age1[500];

int age2[500];

int life_span1[500];
int life_span2[500];
int swap1;
int swap2;
int swap3;
int swap4;

int crossover_mask[34]={ }; // set an binary string
int mutation_mask[34]={ }; // set an binary string

double best11;
double best12;
int best13;
int best14;
int best15;
double best_fitness1;
double total_fitness1;
double average_fitness1;
double parent_average1;
double parent_best1;

double best21;
double best22;
int best23;
int best24;
int best25;
```

```
double best_fitness2;
double total_fitness2;
double average_fitness2;
double parent_average2;
double parent_best2;

int new_gene1[500][34];
int new_gene2[500][34];
int parent_number1[500];
int parent_number2[500];
int parent1[500][34];
int parent2[500][34];
int parent_n1[500];
int parent_n2[500];
int parentnumber1=0;
int parentnumber2=0;
int Chld1[500][34];
int Chld2[500][34];
int age_new_gene1[500];
int age_new_gene2[500];
int new_life_span1[500];
int new_life_span2[500];
int mutate_gene1;
int mutate_gene2;
int alive_number1;
int alive_number2;

double a[2][25]={{-32,-16,0,16,32,-32,-16,0,16,32,-32,-16,0,16,32,-32,-16,0,16,32,-32,-
16,0,16,32},{-32,-32,-32,-32,-32,-16,-16,-16,-16,-
16,0,0,0,0,0,16,16,16,16,16,32,32,32,32,32}};

ofstream fout("out.txt");
ofstream tout1("table1.txt");
ofstream tout2("table2.txt");
ofstream pout1("composition1.txt");
ofstream pout2("composition2.txt");
ofstream rout1("parent1.txt");
ofstream rout2("parent2.txt");

main()
{srand((unsigned)time(NULL));
    initialize();
    g=1;
    printout(g);
    average1(g);
    average2(g);
    for(int i=0; i<40; i++)
    {       if((fit1(i)/average_fitness1)<0.5)
                    life_span1[i]=40;
            else if((fit1(i)/average_fitness1)>=0.5 && (fit1(i)/average_fitness1)<0.55)
```

```
        life_span1[i]=41;
else if((fit1(i)/average_fitness1)>=0.55 && (fit1(i)/average_fitness1)<0.6)
        life_span1[i]=42;
else if((fit1(i)/average_fitness1)>=0.6 && (fit1(i)/average_fitness1)<0.65)
        life_span1[i]=43;
else if((fit1(i)/average_fitness1)>=0.65 && (fit1(i)/average_fitness1)<0.7)
        life_span1[i]=44;
else if((fit1(i)/average_fitness1)>=0.7 && (fit1(i)/average_fitness1)<0.75)
        life_span1[i]=45;
else if((fit1(i)/average_fitness1)>=0.75 && (fit1(i)/average_fitness1)<0.8)
        life_span1[i]=46;
else if((fit1(i)/average_fitness1)>=0.8 && (fit1(i)/average_fitness1)<0.85)
        life_span1[i]=47;
else if((fit1(i)/average_fitness1)>=0.85 && (fit1(i)/average_fitness1)<0.9)
        life_span1[i]=48;
else if((fit1(i)/average_fitness1)>=0.9 && (fit1(i)/average_fitness1)<0.95)
        life_span1[i]=49;
else if((fit1(i)/average_fitness1)>=0.95 && (fit1(i)/average_fitness1)<1.05)
        life_span1[i]=50;
else if((fit1(i)/average_fitness1)>=1.05 && (fit1(i)/average_fitness1)<1.1)
        life_span1[i]=51;
else if((fit1(i)/average_fitness1)>=1.1 && (fit1(i)/average_fitness1)<1.15)
        life_span1[i]=52;
else if((fit1(i)/average_fitness1)>=1.15 && (fit1(i)/average_fitness1)<1.2)
        life_span1[i]=53;
```

```
          else if((fit1(i)/average_fitness1)>=1.2 && (fit1(i)/average_fitness1)<1.25)
                    life_span1[i]=54;
          else if((fit1(i)/average_fitness1)>=1.25 && (fit1(i)/average_fitness1)<1.3)
                    life_span1[i]=55;
          else if((fit1(i)/average_fitness1)>=1.3 && (fit1(i)/average_fitness1)<1.35)
                    life_span1[i]=56;
          else if((fit1(i)/average_fitness1)>=1.35 && (fit1(i)/average_fitness1)<1.4)
                    life_span1[i]=57;
          else if((fit1(i)/average_fitness1)>=1.4 && (fit1(i)/average_fitness1)<1.45)
                    life_span1[i]=58;
          else if((fit1(i)/average_fitness1)>=1.45 && (fit1(i)/average_fitness1)<1.5)
                    life_span1[i]=59;
          else
                    life_span1[i]=60;
}

for(i=0; i<40; i++)
{         if((fit2(i)/average_fitness2)<0.5)
                    life_span2[i]=40;
          else if((fit2(i)/average_fitness2)>=0.5 && (fit2(i)/average_fitness2)<0.55)
                    life_span2[i]=41;
          else if((fit2(i)/average_fitness2)>=0.55 && (fit2(i)/average_fitness2)<0.6)
                    life_span2[i]=42;
          else if((fit2(i)/average_fitness2)>=0.6 && (fit2(i)/average_fitness2)<0.65)
                    life_span2[i]=43;
```

```
else if((fit2(i)/average_fitness2)>=0.65 && (fit2(i)/average_fitness2)<0.7)
        life_span2[i]=44;
else if((fit2(i)/average_fitness2)>=0.7 && (fit2(i)/average_fitness2)<0.75)
        life_span2[i]=45;
else if((fit2(i)/average_fitness2)>=0.75 && (fit2(i)/average_fitness2)<0.8)
        life_span2[i]=46;
else if((fit2(i)/average_fitness2)>=0.8 && (fit2(i)/average_fitness2)<0.85)
        life_span2[i]=47;
else if((fit2(i)/average_fitness2)>=0.85 && (fit2(i)/average_fitness2)<0.9)
        life_span2[i]=48;
else if((fit2(i)/average_fitness2)>=0.9 && (fit2(i)/average_fitness2)<0.95)
        life_span2[i]=49;
else if((fit2(i)/average_fitness2)>=0.95 && (fit2(i)/average_fitness2)<1.05)
        life_span2[i]=50;
else if((fit2(i)/average_fitness2)>=1.05 && (fit2(i)/average_fitness2)<1.1)
        life_span2[i]=51;
else if((fit2(i)/average_fitness2)>=1.1 && (fit2(i)/average_fitness2)<1.15)
        life_span2[i]=52;
else if((fit2(i)/average_fitness2)>=1.15 && (fit2(i)/average_fitness2)<1.2)
        life_span2[i]=53;
else if((fit2(i)/average_fitness2)>=1.2 && (fit2(i)/average_fitness2)<1.25)
        life_span2[i]=54;
else if((fit2(i)/average_fitness2)>=1.25 && (fit2(i)/average_fitness2)<1.3)
        life_span2[i]=55;
else if((fit2(i)/average_fitness2)>=1.3 && (fit2(i)/average_fitness2)<1.35)
```

```
                life_span2[i]=56;
        else if((fit2(i)/average_fitness2)>=1.35 && (fit2(i)/average_fitness2)<1.4)
                life_span2[i]=57;
        else if((fit2(i)/average_fitness2)>=1.4 && (fit2(i)/average_fitness2)<1.45)
                life_span2[i]=58;
        else if((fit2(i)/average_fitness2)>=1.45 && (fit2(i)/average_fitness2)<1.5)
                life_span2[i]=59;
        else
                life_span2[i]=60;
}

do
{       g++;
        if(alive_number1!=0)
                generation1();
        if(alive_number2!=0)
                generation2();

        printout(g);
        average1(g);
        average2(g);
        deletion();
        accident();
        if(g%5==0)
        {       pout1<<"\n"<<g<<":\n";
```

```
                pout2<<"\n"<<g<<":\n";
                for(i=alive_number1-1; i>0; i--)
                {
                        pout1<<age1[i]<<", "<<fit1(i)<<"\n";
                }
                for(i=alive_number2; i>0; i--)
                {
                        pout2<<age2[i]<<", "<<fit2(i)<<"\n";
                }
        }
    }while(g!=500);
    return 0;
}

void initialize()
{   int i,j;
    for(i=0; i<40; i++)
    {       for(j=0; j<34; j++)
            {       individual1[i][j]=initial1[i][j];
                    individual2[i][j]=initial2[i][j];
            }
            age1[i]=initial_age1[i];
            age2[i]=initial_age2[i];
    }
    for(i=0; i<10000; i++)
    {       citizen_number1[i]=i+1;
            citizen_number2[i]=i+1;
            illness1[i]=1;
            illness2[i]=1;
    }}

void alivenumber1(int g)
{   alive_number1=0;
    for(int i=0; bin2dec1(3,i)!=0 && i<500; i++)
    {       alive_number1++;
    }}
void alivenumber2(int g)
{   alive_number2=0;


    for(int i=0; bin2dec2(3,i)!=0 && i<500; i++)
    {alive_number2++;
    }}

void average1(int g)
{   total_fitness1=0;
    average_fitness1=0;
    best_fitness1=fit1(0);
    best11=bin2dec1(1,0);
    best12=bin2dec1(2,0);
    best13=0;
    best14=0;
    best15=0;
    alivenumber1(g);
    for(int i=0; i<alive_number1; i++)
    {       total_fitness1+=(double)fit1(i);
            if(fit1(i)>best_fitness1)
            {       best_fitness1=fit1(i);
                    best11=bin2dec1(1,i);
                    best12=bin2dec1(2,i);
                    best13=citizen_number1[i];
                    best14=illness1[i];
                    best15=age1[i];
            }}
    average_fitness1=(double)total_fitness1/(double)i;
    tout1<< g <<" "<<alive_number1<<" alive, "<<"average_fitness1: ";
    tout1<<(double)average_fitness1;
    tout1<<", best_fitness: "<<best_fitness1;
    tout1<<" ("<<best11<<", "<<best12<<", "<<best15<<") "<<best13;
    tout1<<"\n";
}

void average2(int g)
{   total_fitness2=0;
    average_fitness2=0;
```

```
     best_fitness2=fit2(0);
     best21=bin2dec2(1,0);
     best22=bin2dec2(2,0);
     best23=0;
     best24=0;
     best25=0;
     alivenumber2(g);
     for(int i=0; i<alive_number2; i++)
     {         total_fitness2+=(double)fit2(i);
               if(fit2(i)>best_fitness2)
               {         best_fitness2=fit2(i);
                         best21=bin2dec2(1,i);
                         best22=bin2dec2(2,i);
                         best23=citizen_number2[i];
                         best24=illness2[i];
                         best25=age2[i];
               }}
     average_fitness2=(double)total_fitness2/(double)i;
     tout2<< g <<" "<<alive_number2<<" alive, "<<"total_fitness2: ";
     tout2<<(double)average_fitness2;
     tout2<<", best_fitness: "<<best_fitness2;
     tout2<<" ("<<best21<<", "<<best22<<", "<<best25<<") "<<best23;
     tout2<<"\n";
}
```

```
int test_function()
{    if((val1(best11,best12)==0 && val2(best11,best12)==0 && best11<65.534 &&
best11>(-65.534) && best12<65.534 && best12>(-65.534))
             || (val1(best21,best22)==0 && val2(best21,best22)==0 && best21<65.534
&& best21>(-65.534) && best22<65.534 && best22>(-65.534)))
             return 1;
     else    return 0;
}

double fit1(int i)
{    double fitness=0;
     double x1=bin2dec1(1,i);
     double x2=bin2dec1(2,i);
     for(int j=1; j<26; j++)
     {       fitness+=1/(j+pow((x1-a[0][j-1]),6)+pow((x2-a[1][j-1]),6));
     }
     fitness=(fitness+0.002)/1.002;
     return fitness;
}

double fit2(int i)
{    double fitness=0;
     double x1=bin2dec2(1,i);
     double x2=bin2dec2(2,i);
     for(int j=1; j<26; j++)
```

```
    {       fitness+=1/(j+pow((x1-a[0][j-1]),6)+pow((x2-a[1][j-1]),6));
    }
    fitness=(fitness+0.002)/1.002;
    return fitness;
}

double effective_fitness1(int i)
{   double c;
    if(age1[i]>=25 && age1[i]<life_span1[i])
            c=(double)(1.00);
    else    c=(double)(0);
    return c;
}

double effective_fitness2(int i)
{   double c;
    if(age2[i]>=25 && age2[i]<life_span2[i])
            c=(double)(1.00);
    else    c=(double)(0);
    return c;
}

double val1(double x1,double x2)
{   double value1=0;
    for(int j=1; j<26; j++)
```

```
            value1+=(double)(-6)*(double)pow((double)((double)x1-(double)a[0][j-
1]),5)/(double)pow((double)((double)j+(double)pow((double)((double)x1-
(double)a[0][j-1]),6)+(double)pow((double)((double)x2-(double)a[1][j-1]),6)),2);
    return value1;
}

double val2(double x1,double x2)
{   double value2=0;
    for(int j=1; j<26; j++)
            value2+=(double)(-6)*(double)pow((double)((double)x2-(double)a[1][j-
1]),5)/(double)pow((double)((double)j+(double)pow((double)((double)x1-
(double)a[0][j-1]),6)+(double)pow((double)((double)x2-(double)a[1][j-1]),6)),2);
    return value2;
}

double bin2dec1(int m,int i)
{   double dec=0;
    int j;
    if(m==1)
    {       for(j=1; j<17; j++)
                    (double)dec+=(double)(individual1[i][j])*(double)pow(2.0,(16-j));
            if(individual1[i][0]==1)
                    return (double)(-1)*dec/1000;
            else return (double)dec/1000;
    }
    else if(m==2)
    {       for(j=1; j<17; j++)
                    (double)dec+=(double)(individual1[i][j+17])*(double)pow(2.0,(16-
j));
            if(individual1[i][17]==1)
                    return (double)(-1)*dec/1000;
            else return (double)dec/1000;
    }
    else
    {       for(j=0; j<34; j++)
                    (double)dec+=(double)(individual1[i][j])*(double)pow(2.0,(33-j));
            return (double)dec/1000;
    }}
double bin2dec2(int m,int i)
{   double dec=0;
    int j;
    if(m==1)
    {       for(j=1; j<17; j++)
                    (double)dec+=(double)(individual2[i][j])*(double)pow(2.0,(16-j));
            if(individual2[i][0]==1)
                    return (double)(-1)*dec/1000;
            else return (double)dec/1000;
    }
    else if(m==2)
    {       for(j=1; j<17; j++)
```

```
                (double)dec+=(double)(individual2[i][j+17])*(double)pow(2.0,(16-
j));
            if(individual2[i][17]==1)
                return (double)(-1)*dec/1000;
            else return (double)dec/1000;
    }
    else
    {       for(j=0; j<34; j++)
                (double)dec+=(double)(individual2[i][j])*(double)pow(2.0,(33-j));
            return (double)dec/1000;
    }}

double bin2dec12(int i)
{   double dec=0;
    int j;
    for(j=0; j<34; j++)
            (double)dec+=(double)(new_gene1[i][j])*(double)pow(2.0,(33-j));
    return (double)dec/1000;
}
double bin2dec22(int i)
{   double dec=0;
    int j;
    for(j=0; j<34; j++)
            (double)dec+=(double)(new_gene2[i][j])*(double)pow(2.0,(33-j));
    return (double)dec/1000;
```

```
}

void printout(int g)
{    cout<<g<<": \n";
     fout<<g<<": \n";
}

void generation1()
{    re_initial1();
     double f1=child_number1();
     if(f1==0)
     {          new_generation1(0);
                new_individual1();
     }
     else if(f1==(-1))
     {          for(int i=0; i<500; i++)
                          for(int j=0; j<34; j++)
                                    individual1[i][j]=0;
     }
     else
     {          select_parent1(f1);
                crossover1(f1);
                new_generation1(f1);
                new_individual1();
     }}
```

```
void generation2()
{    re_initial2();
     double f2=child_number2();
     if(f2==0)
     {          new_generation2(0);
                new_individual2();
     }
     else if(f2==(-1))
     {          for(int i=0; i<500; i++)
                          for(int j=0; j<34; j++)
                                    individual2[i][j]=0;
     }
     else
     {          select_parent2(f2);
                crossover2(f2);
                new_generation2(f2);
                new_individual2();
     }}

void re_initial1()
{    parentnumber1=0;
     parent_average2=0;
     parent_best1=0;
     for(int i=0; i<500; i++)
```

```
        {       parent_number1[i]=0;
                parent_n1[i]=0;
                mutate_gene1=0;
                age_new_gene1[i]=0;
                new_life_span1[i]=0;
                for(int j=0; j<34; j++)
                {       Chld1[i][j]=0;
                        new_gene1[i][j]=0;
                }}}
void re_initial2()
{   parentnumber2=0;
    parent_average2=0;
    parent_best2=0;
    for(int i=0; i<500; i++)
    {       parent_number2[i]=0;
            parent_n2[i]=0;
            mutate_gene2=0;
            age_new_gene2[i]=0;
            new_life_span2[i]=0;
            for(int j=0; j<34; j++)
            {       Chld2[i][j]=0;
                    new_gene2[i][j]=0;
            }}}
```

```
double child_number1()
{    int f;
     for(int c=0; c<alive_number1; c++)
     {         if(effective_fitness1(c)>0)
               {         parent_number1[parentnumber1]=c;
                         parentnumber1++;
                         parent_average1+=fit1(c);
                         if(fit1(c)>parent_best1)
                                   parent_best1=fit1(c);
               }}
     parent_average1=parent_average1/parentnumber1;
     rout1<<g<<" : "<<"parent number is: "<<parentnumber1<<", parent best is:
"<<parent_best1<<", average is: "<<parent_average1<<"\n";
     if(alive_number1==0)    f=(-1);
     else if(parentnumber1==0)         f=0;
     else      f=(int)((double)0.04*(double)alive_number1);
     return f;
}

double child_number2()
{    int f;
     for(int c=0; c<alive_number2; c++)
     {         if(effective_fitness2(c)>0)
               {         parent_number2[parentnumber2]=c;
                         parentnumber2++;
```

```
                        parent_average2+=fit2(c);
                        if(fit2(c)>parent_best2)
                                parent_best2=fit2(c);
                }}
        parent_average2=parent_average2/parentnumber2;
        rout2<<g<<" : "<<"parent number is: "<<parentnumber2<<", parent best is:
"<<parent_best2<<", average is: "<<parent_average2<<"\n";

        if(alive_number1==0)    f=(-1);
        else if(parentnumber2==0)         f=0;
        else        f=(int)((double)0.04*(double)alive_number2);
        return f;
}

void select_parent1(int f)
{       int m,p;
        int n,q;
        //srand((int)time(0))
        for(int i=0; i<f; i++)
        {           m=rand()%parentnumber1;
                    n=rand()%parentnumber1;
                    p=rand()%parentnumber1;
                    q=rand()%parentnumber1;
                    if(fit1(parent_number1[m])>fit1(parent_number1[n]))
                    {           for(int j=0; j<34; j++)
                    {       parent1[2*i][j]=individual1[parent_number1[m]][j];
                    }
                    fout<<"1 selected parents is:
"<<citizen_number1[parent_number1[m]]<<"("<<age1[parent_number1[m]]
   <<","<<fit1(parent_number1[m])<<")"<<"\n";
            }
            else
            {       for(int j=0; j<34; j++)
                    {       parent1[2*i][j]=individual1[parent_number1[n]][j];
                    }
                    fout<<"1 selected parents are:
"<<citizen_number1[parent_number1[n]]<<"("<<age1[parent_number1[n]]<<","<<fit1(
parent_number1[n]) <<")"<<"\n";
            }
            if(fit1(parent_number1[p])>fit1(parent_number1[q]))
            {       for(int j=0; j<34; j++)
                    {       parent1[2*i+1][j]=individual1[parent_number1[p]][j];
                    }
                    fout<<"1 selected parents is:
"<<citizen_number1[parent_number1[p]]<<"("<<age1[parent_number1[p]]<<","<<fit1(
parent_number1[p]) <<")"<<"\n";
            }
            else
            {       for(int j=0; j<34; j++)
                    {       parent1[2*i+1][j]=individual1[parent_number1[q]][j];
                    }
                    fout<<"1 selected parents are:
"<<citizen_number1[parent_number1[q]]<<"("<<age1[parent_number1[q]]<<","<<fit1(
parent_number1[q])<<")"<<"\n";
          }}}

void select_parent2(int f)
{   int m,p;
    int n,q;
    //srand((int)time(0))
    for(int i=0; i<f; i++)
    {         m=rand()%parentnumber2;
              n=rand()%parentnumber2;
              p=rand()%parentnumber2;
              q=rand()%parentnumber2;
              if(fit2(parent_number2[m])>fit2(parent_number2[n]))
              {         for(int j=0; j<34; j++)
                        {         parent2[2*i][j]=individual2[parent_number2[m]][j];
                        }
                        fout<<"2 selected parents is:
"<<citizen_number2[parent_number2[m]]<<"("<<age2[parent_number2[m]]<<","<<fit
2(parent_number2[m])<<")"<<"\n";
              }
              else
              {         for(int j=0; j<34; j++)
```

```
                {parent2[2*i][j]=individual2[parent_number2[n]][j];
                }
                fout<<"2 selected parents are:
"<<citizen_number2[parent_number2[n]]<<"("<<age2[parent_number2[n]]<<","<<fit2(
parent_number2[n])<<")"<<"\n";
        }
        if(fit2(parent_number2[p])>fit2(parent_number2[q]))
        {       for(int j=0; j<34; j++)
                {       parent2[2*i+1][j]=individual2[parent_number2[p]][j];
                }
                fout<<"2 selected parents is:
"<<citizen_number2[parent_number2[p]]<<"("<<age2[parent_number2[p]]<<","<<fit2(
parent_number2[p])<<")"<<"\n";
        }
        else
        {       for(int j=0; j<34; j++)
                {       parent2[2*i+1][j]=individual2[parent_number2[q]][j];
                }
                fout<<"2 selected parents are:
"<<citizen_number2[parent_number2[q]]<<"("<<age2[parent_number2[q]]<<","<<fit2(
parent_number2[q])<<")"<<"\n";
        }}}

void crossover1(int f)
{   for(int i=0; i<f; i++)
```

```
    {        int m=rand()%34;
             for(int j=0; j<m; j++)
             {        individual1[499][j]=parent1[2*i+1][j];
                      individual1[498][j]=parent1[2*i][j];
             }
             for(j=m; j<34; j++)
             {        individual1[499][j]=parent1[2*i][j];
                      individual1[498][j]=parent1[2*i+1][j];
             }
             if(fit1(499)>fit1(498))
             {        for(int j=0; j<34; j++)
                               Chld1[i][j]=individual1[499][j];
             }
             else
             {        for(int j=0; j<34; j++)
                               Chld1[i][j]=individual1[498][j];
             }
             for(j=0;j<34;j++)
             {        individual1[499][j]=0;
                      individual1[498][j]=0;
             }}}

void crossover2(int f)
{    for(int i=0; i<f; i++)
    {        int m=rand()%34;
```

```
                for(int j=0; j<m; j++)
                {       individual2[499][j]=parent2[2*i+1][j];
                        individual2[498][j]=parent2[2*i][j];
                }
                for(j=m; j<34; j++)
                {       individual2[499][j]=parent2[2*i][j];
                        individual2[498][j]=parent2[2*i+1][j];
                }
                if(fit2(499)>fit2(498))
                {       for(int j=0; j<34; j++)
                                Chld2[i][j]=individual2[499][j];
                }
                else
                {       for(int j=0; j<34; j++)
                                Chld2[i][j]=individual2[498][j];
                }
                for(j=0;j<34;j++)
                {       individual2[499][j]=0;
                        individual2[498][j]=0;
                }}}
void new_generation1(int f)
{   int i,j;
    for(i=0; i<alive_number1; i++)
    {           for(j=0; j<34; j++)
                {new_gene1[i][j]=individual1[i][j];
```

```
            }
            age_new_gene1[i]=age1[i]+1;
            new_life_span1[i]=life_span1[i];
    }
    for(int m=0; m<f; m++)
    {       for(j=0; j<34; j++)
            {       new_gene1[i][j]=Chld1[m][j];    }
            age_new_gene1[i]=1;
            new_life_span1[i]=100;
            mutation1(i);
            i++;
    }}

void new_generation2(int f)
{   int i,j;
    for(i=0; i<alive_number2; i++)
    {       for(j=0; j<34; j++)
            {       new_gene2[i][j]=individual2[i][j];      }
            age_new_gene2[i]=age2[i]+1;
            new_life_span2[i]=life_span2[i];
    }
    for(int m=0; m<f; m++)
    {       for(j=0; j<34; j++)
            {new_gene2[i][j]=Chld2[m][j];  }
            age_new_gene2[i]=1;
```

```
                new_life_span2[i]=100;
                mutation2(i);
                i++;
        }}

void mutation1(int i)
{int j,b;
        //srand((int)time(0))
        b = rand() % 1000;
        if(b<=20)
        {       j=rand()%34;
                new_gene1[i][j]=abs(new_gene1[i][j]-1);
                fout<<"\n"<<"mutation individual in p1: "<<citizen_number1[i]<<"\n";
        }}

void mutation2(int i)
{   int j,b;
        //srand((int)time(0))
        b = rand() % 1000;
        if(b<=20)
        {       j=rand()%34;
                new_gene2[i][j]=abs(new_gene2[i][j]-1);
                fout<<"\n"<<"mutation individual in p2: "<<citizen_number2[i]<<"\n";
        }}
```

```
void new_individual1()
{    int i, j,m;
     for(i=0;i<500;i++)
     {          for(j=0;j<34;j++)
                         individual1[i][j]=0;
                age1[i]=0;
                life_span1[i]=0;
     }
     for(i=0,m=0; i<500; i++,m++)
     {          if(age_new_gene1[m]<=new_life_span1[m] && age_new_gene1[m]!=0 )
                {          for(j=0; j<34; j++)
                           {          individual1[i][j]=new_gene1[m][j];                    }
                           age1[i]=age_new_gene1[m];
                           life_span1[i]=new_life_span1[m];
                }
                else if(age_new_gene1[m]>new_life_span1[m])
                {          fout<<"\n-----------------------dead citizen 1:
"<<citizen_number1[i]<<", age: "<<age_new_gene1[m]<<", "<<new_life_span1[m];
                           for(int p=i;p<10000; p++)
                           {          citizen_number1[p]=citizen_number1[p+1];
                                      illness1[p]=illness1[p+1];
                           }
                           i=i-1;
                }
                else if(age_new_gene1[m]==0)
```

```
                        break;
        }


        fout<<"\n";
        for(i=0;i<500;i++)
                if(age1[i]==1) fout<<"1 Child is: "<<citizen_number1[i]<<"\n";
}


void new_individual2()
{       int i, j,m;
        for(i=0;i<500;i++)
        {       for(j=0;j<34;j++)
                        individual2[i][j]=0;
                age2[i]=0;
                life_span2[i]=0;
        }
        for(i=0,m=0; i<500; i++,m++)
        {       if(age_new_gene2[m]<=new_life_span2[m] && age_new_gene2[m]>0)
                {       for(j=0; j<34; j++)
                        {       individual2[i][j]=new_gene2[m][j];      }
                        age2[i]=age_new_gene2[m];
                        life_span2[i]=new_life_span2[m];
                }
                else if(age_new_gene2[m]>new_life_span2[m])
```

```
		{	fout<<"\n----------------------dead citizen 2:
"<<citizen_number2[i]<<", age: "<<age_new_gene2[m]<<", "<<new_life_span2[m];
			for(int p=i;p<10000; p++)
			{	citizen_number2[p]=citizen_number2[p+1];
				illness1[p]=illness1[p+1];
			}
			i=i-1;
		}
		else if(age_new_gene2[m]==0)
			break;
	}

	fout<<"\n";
	for(i=0;i<500;i++)
		if(age2[i]==1) fout<<"2 Child is: "<<citizen_number2[i]<<"\n";
}

void deletion()
{	int i,m;
	int b;
	int total_number;
	for(i=0; i<500; i++)
	{	fit[i]=0;
		sort_number[i]=0;
		if(life_span1[i]==100)
```

```
          {          if((fit1(i)/average_fitness1)<0.5)
                          life_span1[i]=40;
                     else if((fit1(i)/average_fitness1)>=0.5 &&
(fit1(i)/average_fitness1)<0.55)
                          life_span1[i]=41;
                     else if((fit1(i)/average_fitness1)>=0.55 &&
(fit1(i)/average_fitness1)<0.6)
                          life_span1[i]=42;
                     else if((fit1(i)/average_fitness1)>=0.6 &&
(fit1(i)/average_fitness1)<0.65)
                                          life_span1[i]=43;
                     else if((fit1(i)/average_fitness1)>=0.65 &&
(fit1(i)/average_fitness1)<0.7)
                          life_span1[i]=44;
                     else if((fit1(i)/average_fitness1)>=0.7 &&
(fit1(i)/average_fitness1)<0.75)
                          life_span1[i]=45;
                     else if((fit1(i)/average_fitness1)>=0.75 &&
(fit1(i)/average_fitness1)<0.8)
                          life_span1[i]=46;
                     else if((fit1(i)/average_fitness1)>=0.8 &&
(fit1(i)/average_fitness1)<0.85)
                          life_span1[i]=47;
                     else if((fit1(i)/average_fitness1)>=0.85 &&
(fit1(i)/average_fitness1)<0.9)
```

```
                        life_span1[i]=48;
                else if((fit1(i)/average_fitness1)>=0.9 &&
(fit1(i)/average_fitness1)<0.95)
                        life_span1[i]=49;
                else if((fit1(i)/average_fitness1)>=0.95 &&
(fit1(i)/average_fitness1)<1.05)
                        life_span1[i]=50;
                else if((fit1(i)/average_fitness1)>=1.05 &&
(fit1(i)/average_fitness1)<1.1)
                        life_span1[i]=51;
                else if((fit1(i)/average_fitness1)>=1.1 &&
(fit1(i)/average_fitness1)<1.15)
                        life_span1[i]=52;
                else if((fit1(i)/average_fitness1)>=1.15 &&
(fit1(i)/average_fitness1)<1.2)
                        life_span1[i]=53;
                else if((fit1(i)/average_fitness1)>=1.2 &&
(fit1(i)/average_fitness1)<1.25)
                        life_span1[i]=54;
                else if((fit1(i)/average_fitness1)>=1.25 &&
(fit1(i)/average_fitness1)<1.3)
                        life_span1[i]=55;
                else if((fit1(i)/average_fitness1)>=1.3 &&
(fit1(i)/average_fitness1)<1.35)
                        life_span1[i]=56;
```

```
                    else if((fit1(i)/average_fitness1)>=1.35 &&
(fit1(i)/average_fitness1)<1.4)
                              life_span1[i]=57;
                    else if((fit1(i)/average_fitness1)>=1.4 &&
(fit1(i)/average_fitness1)<1.45)
                              life_span1[i]=58;
                    else if((fit1(i)/average_fitness1)>=1.45 &&
(fit1(i)/average_fitness1)<1.5)
                              life_span1[i]=59;
                    else
                              life_span1[i]=60;
          }}

     if(life_span2[i]==100)
     {if((fit2(i)/average_fitness2)<0.5)
          life_span2[i]=40;
     else if((fit2(i)/average_fitness2)>=0.5 && (fit2(i)/average_fitness2)<0.55)
          life_span2[i]=41;
     else if((fit2(i)/average_fitness2)>=0.55 && (fit2(i)/average_fitness2)<0.6)
          life_span2[i]=42;
     else if((fit2(i)/average_fitness2)>=0.6 && (fit2(i)/average_fitness2)<0.65)
          life_span2[i]=43;
     else if((fit2(i)/average_fitness2)>=0.65 && (fit2(i)/average_fitness2)<0.7)
          life_span2[i]=44;
     else if((fit2(i)/average_fitness2)>=0.7 && (fit2(i)/average_fitness2)<0.75)
```

```
        life_span2[i]=45;
else if((fit2(i)/average_fitness2)>=0.75 && (fit2(i)/average_fitness2)<0.8)
        life_span2[i]=46;
else if((fit2(i)/average_fitness2)>=0.8 && (fit2(i)/average_fitness2)<0.85)
        life_span2[i]=47;
else if((fit2(i)/average_fitness2)>=0.85 && (fit2(i)/average_fitness2)<0.9)
        life_span2[i]=48;
else if((fit2(i)/average_fitness2)>=0.9 && (fit2(i)/average_fitness2)<0.95)
        life_span2[i]=49;
else if((fit2(i)/average_fitness2)>=0.95 && (fit2(i)/average_fitness2)<1.05)
        life_span2[i]=50;
else if((fit2(i)/average_fitness2)>=1.05 && (fit2(i)/average_fitness2)<1.1)
        life_span2[i]=51;
else if((fit2(i)/average_fitness2)>=1.1 && (fit2(i)/average_fitness2)<1.15)
        life_span2[i]=52;
else if((fit2(i)/average_fitness2)>=1.15 && (fit2(i)/average_fitness2)<1.2)
        life_span2[i]=53;
else if((fit2(i)/average_fitness2)>=1.2 && (fit2(i)/average_fitness2)<1.25)
        life_span2[i]=54;
else if((fit2(i)/average_fitness2)>=1.25 && (fit2(i)/average_fitness2)<1.3)
        life_span2[i]=55;
else if((fit2(i)/average_fitness2)>=1.3 && (fit2(i)/average_fitness2)<1.35)
        life_span2[i]=56;
else if((fit2(i)/average_fitness2)>=1.35 && (fit2(i)/average_fitness2)<1.4)
        life_span2[i]=57;
```

```
                else if((fit2(i)/average_fitness2)>=1.4 && (fit2(i)/average_fitness2)<1.45)
                        life_span2[i]=58;
                else if((fit2(i)/average_fitness2)>=1.45 && (fit2(i)/average_fitness2)<1.5)
                        life_span2[i]=59;
                else
                        life_span2[i]=60;
                }}

        total_number=alive_number1+alive_number2;

       if(total_number>120)
        {       for(i=0; i<=alive_number1; i++)
                {       fit[i]=fit1(i);
                        sort_number[i]=i*10+1;
                }
                for(m=0; m<alive_number2; m++,i++)
                {       fit[i]=fit2(m);
                        sort_number[i]=m*10+2;
                }
                sort(total_number);
                for(i=total_number; i>120; i--)
                {       if(sort_number[i]%2==1)
                        {       b=(sort_number[i]-1)/10;
                                fout<<"\n----------------------illness citizen 1:
"<<citizen_number1[b];
```

```
                    illness1[b]=illness1[b]+0.5;
                    fout<<", illness1 is: "<<illness1[b];
                    fout<<"\n";
                }
                else if(sort_number[i]%2==0)
                {   b=(sort_number[i]-2)/10;
                    fout<<"\n----------------------illness citizen 2:
"<<citizen_number2[b];
                    illness2[b]=illness2[b]+0.5;
                    fout<<", illness2 is: "<<illness2[b];
                    fout<<"\n";
                }}}}

void sort(int n)
{  int i,j;
   for(i=0; i<n; i++)
   {     for(j=i+1; j<n; j++)
         {     if(fit[j]>fit[i])
               {     swap1=fit[i];
                     fit[i]=fit[j];
                     fit[j]=swap1;
                     swap2=sort_number[i];
                     sort_number[i]=sort_number[j];
                     sort_number[j]=swap2;
               }}}}
```

```
void accident()
{    int i,j,m,n,b;
     for(i=0,m=0; m<alive_number1; m++,i++)
     {          b = rand()%1000;
                if(b<2*illness1[i])
                {          fout<<"\n----------------------accident 1: "<<citizen_number1[i]<<",
age: "<<age1[i]<<", illness1: "<<illness1[i]<<", b: "<<b<<"\n";
                           for(n=i; n<alive_number1; n++)
                           {          for(j=0; j<34; j++)
                                                 individual1[n][j]=individual1[n+1][j];
                                      age1[n]=age1[n+1];
                           }
                           for(j=0; j<34; j++)         individual1[n][j]=0;
                           age1[n]=0;
                           for(int p=i;p<10000; p++)
                           {          citizen_number1[p]=citizen_number1[p+1];
                                      illness1[p]=illness1[p+1];
                           }
                           i=i-1;
                }}

     for(i=0,m=0; m<alive_number2; m++,i++)
     {          b = rand()%1000;
                if(b<2*illness2[i])
```

```
	{	fout<<"\n----------------------accident 2: "<<citizen_number2[i]<<", 
age: "<<age2[i]<<", illness2: "<<illness2[i]<<", b: "<<b<<"\n";
		for(n=i; n<alive_number2+1; n++)
		{for(j=0; j<34; j++) individual2[n][j]=individual2[n+1][j];
		age2[n]=age2[n+1];
		}
		for(j=0; j<34; j++)		individual2[n][j]=0;
		age2[n]=0;
		for(int p=i;p<10000; p++)
		{	citizen_number2[p]=citizen_number2[p+1];
			illness2[p]=illness2[p+1];
		}
		i=i-1;
	}}}
```