4.1 Introduction

Genetic Algorithms are search algorithm based on mechanics of natural genetics. They are based on operations existing in nature. They combine a Darwinian survival of the fittest approach with a structured, yet randomized, information exchange. The advantage is they can search complex and large amount of spaces efficiently and locate near optimal solutions pretty rapidly.

Solution to a problem solved by genetic algorithm uses an evolutionary process (it is evolved). Algorithm begins with a **set of solutions** (represented by **chromosomes**) called **population**. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are then selected to form new solutions (**offspring**) are selected according to their fitness - the more suitable they are the more chances they have to reproduce. This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied [5].

4.2 Biological Terminology

At this point it is useful to formally introduce some of the biological terminology that will be used throughout the book. In the context of genetic algorithms, these biological terms are used in the spirit so analogy with real biology, through the entities they refer to are much simpler than the real biological ones.

All living organisms consist of cells, and each cell contains the same set of one or more chromosomes – strings of DNA – that serve as a "blueprint" for the algorithms. A chromosome can be conceptually divided into genes – functional blocks of DNA, each of which encodes a particular protein. Very roughly, one can think of a gene as encoding a trail, such as eye

color. The different possible "settings" for a trail (e.g., blue, brown, hazel) are called alleles. Each gene is located at a particular locus (position) on the chromosome.

Many organisms have multiple chromosomes in each cell. The complete collection of genetic material (all chromosomes taken together) is called the organism's genome. The term genotype refers to the particular set of genes contained in a genome. Two individuals that have identical genomes are said to have the same genotype. The genotype gives rise, under fetal and later development, to the organism's phenotype – its physical and mental characteristics, such as eye color, height, brain size, and intelligence [11].

4.3 How is GA Different ?

- GAs work on " Encoding "of the parameter (Not the parameters!).
- GAs search from a population of points (Implicit Parallelism).
- GAs use pay-off information only. They don't require any extraneous information.
- GAs use probabilistic transition rules, not deterministic rules [3].

4.4 Basic Genetic Algorithm

The Basic Genetic Algorithm procedure is [10]:

```
Procedure Genetic algorithm
 Begin
   t := 0
  Initialize Population (t)
 Evaluate Population (t)
 While (not termination-condition) do
   Begin
    t  := t + 1
    Select Population (t) from Population (t - 1)
   Alter Population (t)
    Evaluate Population (t)
   End
End
```

To solve any problem using GA, we should consider the following:

1. Encoding.
2. Initialization
3. Selection
4. Alteration [3].

The important elements are the encoding of chromosomes, the initialization of the population, evaluation of each individual in the

population, selection of those individuals to pass on to the next generation, and alteration of the selected individuals to generate new, hopefully better, solutions. We will examine each of these elements in turn, however decisions made in one portion of the algorithm could affect choices available elsewhere [10].

4.5 Encoding

Encoding of chromosomes is the first question to ask when starting to solve a problem with GA. Encoding depends on the problem heavily. We introduced some encoding that has been already used with some success.

1-Binary Encoding

Binary encoding is the most common one, mainly because the first research of GA used this type of encoding and because of its relative simplicity.

In **binary encoding**, every chromosome is a string of **bits** (0 or **1)** as shown in figure (4.1).

Chromosome A	**1011001011001010111100101**
Chromosome B	**1111111000001100000111111**

Figure(4.1)Example of chromosomes with binary encoding

Binary encoding gives many possible chromosomes even with a small number of alleles. On the other hand, this encoding is often not natural for many problems and sometimes corrections must be made after crossover and/or mutation.

Example of Problem: ***Knapsack problem***

The problem: There are things with given value and size. The knapsack has given capacity. Select things to maximize the value of things in knapsack, but do not extend knapsack capacity.

Encoding : Each bit says, whether the corresponding thing is in knapsack.

2-Permutation Encoding

Permutation encoding can be used in ordering problems, such as travelling salesman problem or task ordering problem.

In **permutation encoding**, every chromosome is a string of numbers that represent a position in a **sequence** as shown in figure(4.2).

Chromosome A	**1 5 3 2 6 4 7 9 8**
Chromosome B	**8 5 6 7 2 3 1 4 9**

Figure(4.2) Example of chromosomes with permutation encoding

Permutation encoding is useful for ordering problems. For some types of crossover and mutation corrections must be made to leave the chromosome consistent (i.e. have real sequence in it) for some problems.

Example of Problem: Travelling salesman problem (TSP)

The problem: There are cities and given distances between them. Travelling salesman has to visit all of them, but he does not want to travel more than necessary. Find a sequence of cities with a minimal traveled distance.

Encoding : Chromosome describes the order of cities, in which the salesman will visit them.

3-Value Encoding

Direct value encoding can be used in problems where some more complicated values such as real numbers are used. Use of binary encoding for this type of problems would be difficult.

In the **value encoding**, every chromosome is a sequence of some values. Values can be anything connected to the problem, such as (real) numbers, chars or any objects as shown in figure (4.3).

Chromosome A	1.2324 5.3243 0.4556 2.3293 2.4545
Chromosome B	ABDJEIFJDHDIERJFDLDFLFEGT
Chromosome C	(back), (back), (right), (forward), (left)

Figure (4.3) Example of chromosomes with value encoding

Value encoding is a good choice for some special problems. However, for this encoding it is often necessary to develop some new crossover and mutation specific for the problem.

Example of Problem: ***Finding weights for a neural network***

The problem: A neural network is given with defined architecture. Find weights between neurons in the neural network to get the desired output from the network.

Encoding : Real values in chromosomes represent weights in the neural network.

4-Tree Encoding

Tree encoding is used mainly for evolving programs or expressions, i.e. for **genetic programming**.

In the **tree encoding** every chromosome is a tree of some objects, such as functions or commands in programming language.

Tree encoding is useful for evolving programs or any other structures that can be encoded in trees. Programming language LISP is often used for this purpose, since programs in LISP are represented directly in the form of tree and can be easily parsed as a tree, so the crossover and mutation can be done relatively easily.

Example of Problem: ***Finding a function that would approximate given pairs of values***

The problem: Input and output values are given. The task is to find a function that will give the best outputs (i.e. the closest to the wanted ones) for all inputs.

Encoding: Chromosome are functions represented in a tree [5].

4.6 Initialization

First, we must determine how large to make the population. Again, it is a matter of choice that should be tuned to the specific problem at hand. We wish to initialize the population with diverse individuals. Why? Because they will be learning from each other. One of the issues GA have to deal with is premature convergence to sub-optimal solutions due to a lack of diversity in the population. There are many ways to arrange this initial diversity:

1-Uniformly Random

We create tours randomly from the search space with a uniform distribution. But this is not the only way.

2-Grid initialization

Here, we seed the population with selection from regular intervals in the search space. The size of the intervals, and exactly what makes an interval, is generally problem dependent.

3-Non-clustering

Another restriction we can place on our population to ensure diversity is a non-clustering rule. Each newly generated individual must be a predefined distance away from all previously added individuals. For TSP, we could say that all new individuals must be at least 2 applications of 2-swap away from each other.

4-*Local Optimization*

A final method for initialization is to use the solutions found by other search techniques, such as hill-climbing or SA. While this does not encourage diversity, we can guarantee that our genetic algorithm will do at least as well as the initial seed algorithm, and this can help reassure some skeptics.

4.7 Evaluation Functions

Selecting which individuals can reproduce is based on the evaluation and comparison of solutions. Therefore we must take care when designing an evaluation function, such that it discriminates between

better and worse solutions. For example, let's say our task is to find the string of text "HAPPY" and our possible solutions are any arrangement of 5 alphabetic characters. The size of this search space is 5^{26}. How can we distinguish between good and bad strings? One possible, albeit very poor, function assigns the word "HAPPY" a score of 1, and all other strings a score of 0. You can see how hard it would be to search using this evaluation function. To develop a good evaluation function, one thing we can do is look at our operators. Suppose we have an operator that lets us replace a particular character with another one, similar to 1-flip in SAT. A useful evaluation function would be based on how many letters of "HAPPY" are in the correct position. For instance, "HXOPN" receives score of 2, while "OOGIE" receives a score of 0. But under this evaluation function, the string "APPYH" would receive a score of 0, even though all the letters of the goal string are present. With only our 1-flip operator, this is reasonable, but suppose we add another operator, that allows us to rotate the string either left or right. Now we would like "APPYH" to receive a much better score than 0. A more complex evaluation function we could derive would be 1 point for each letter of the string that is a part of the goal string, and an additional point for each letter in the correct position. Then "HXOPN" would still receive a score of 2, while "APPYH" is now evaluated to be 6, and "HAPXY" would have a score of 8.

4.8 Selection

Selection of individuals for the next generation, either to reproduce or to live on, relies heavily on the evaluation function above. How heavily is dependent on which selection technique you use. We wish to apply

some pressure so that good solutions survive, and weak solutions die; too much, and we converge to less than optimal solutions, too little and we never make progress towards the solution. Again, it is a balancing act to find the right selection technique for the problem at hand.

1-Deterministic

In deterministic selection, only the best survive. This leads to very fast vengeance. Two deterministic selection techniques are common, one that includes parents in determining the best solutions, and one which replaces all parents with children.

We can represent the size of the population as "mu", and the number of children generated as "lambda". (mu + lambda) selection chooses the best "mu" to continue to the next generation, and the competition is between both parents and children. (mu, lambda) selection again chooses the best "mu", however it is only the children that factor into who's the best. Parents are thrown away to fight early convergence. Deterministic selection relies very heavily on the evaluation function, and converges the fastest of all methods we will discuss.

2-Proportional Fitness

Instead of taking the best "mu", each individual can be selected proportionally to their evaluation score. Suppose we have the following population:

Individual	Score
A	4
B	10
C	14
D	7
E	9
F	6

Figure (4.4) Hypothetical Population

The sum of their scores is 50. This gives individual A 8% chance of being selected, individual B 20%, etc. We usually implement this with what is called "roulette wheel selection". Select a random number between 0 and 1. Then progressively add on the probabilities of each individual in order, until this sum is greater than the random number. For example, I randomly choose 0.77. This selects individual E, since 8 + 20 + 28 + 14 + 18 is 88%. With a random choice of 0.34, we select C, since 8 + 20 + 28 = 56%. To select the next generation, we would need to choose "mu" random numbers.

We see in proportional fitness, even the worst individual, A has a chance to reproduce, albeit only 8%. This will help prevent stagnation in the population.

3-Tournament Selection

In this scheme, two individuals are selected at random with replacement from the population, and the one with the best score gets selected to reproduce. Using the above example, one round of tournament

selection could choose B and D for competition. B would then be selected since its score of 10 is larger than D's score of 7. Repeat this "mu" times to get the next population.

So how is this different from proportional fitness? Now, A has a 1/36 chance of reproducing (the chance of choosing A for both sides of the competition), about 2.8%, while C, the most fit individual, has a 11/36 chance, or 30.6%. Tournament selection does not care about the spread of the scores, only the ranking. The nth ranked individual in a population of size mu will have a $(2mu - 2n + 1) / mu^2$ chance of reproducing. This puts an upper and lower bound on the chances of any individual to reproduce for the next generation. Tournament selection can be generalized to include more than 2 individuals being chosen for competition, and selecting the best from this group [10].

4.9 Alteration

There are a number of genetic operators that are used to alter the population:

1-Mutation

The first most basic way to alter a solution for the next generation is to use mutation [10]. Mutation plays a decidedly secondary role in the operation of GA. Mutation is needed because, even though reproduction and crossover effectively search and recombine extant notions, occasionally they may lose some potentially useful genetic material.

The simple mutation method by Holland (1975), flips a random bit on or off. There are several types of Mutation such as “Shift change” mutation, and “Exchange” mutation. In Shift mutation, a column of

chromosome at one position is removed and at another position as shown in figure (4.5-a). The two positions are randomly selected, the column 6 of child 1 is removed and inserted between columns 2 and 3.

Exchange mutation, randomly selects two positions in a given chromosome and exchange both genes. The remaining genes are kept intact as shown in figure (4.5-b) [12].

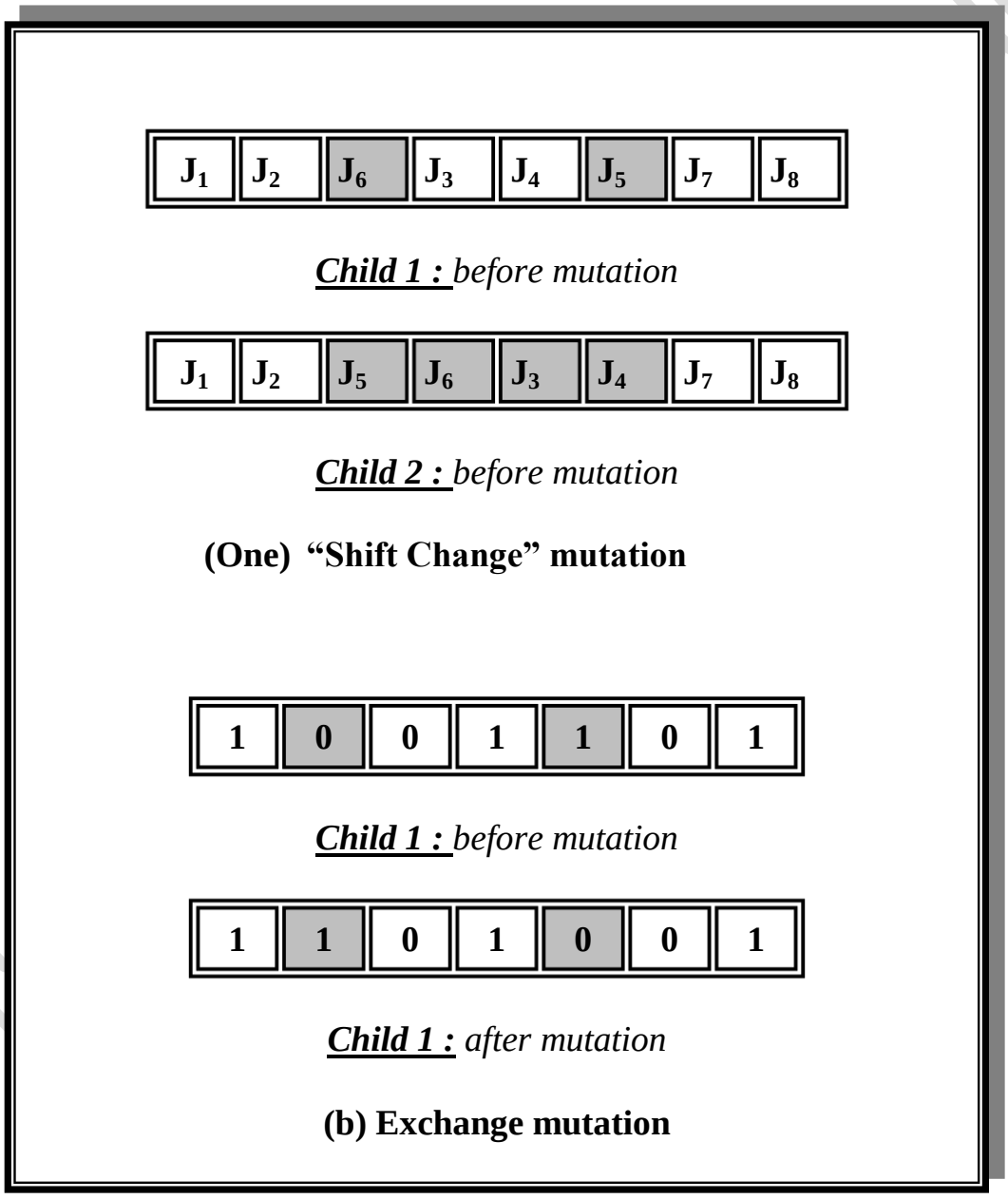

Figure (4.5) GA – Mutation

2-Crossover

But the interesting behavior arises from genetic algorithms because of the ability of solutions to learn from each other. Solutions can combine to form offspring for the next generation. Sometimes they will pass on their worst information, but if we do crossover in combination with a forceful selection technique, then we should see better solutions result. Since there are many details to crossover with permutations as in TSP, we will cover the basic crossover techniques, known as "cut and splice" techniques, for vectors today, such as SAT.

- ***One-point***

We select two individuals to be parents for the next generation, and choose some point along the vector, between 0 and the length of the vector. This will be our crossover point between the two parents. We swap information after the crossover point to make our two new children. For example, we have

1101101101 and 0001001000

as our two parents, and choose the crossover point to be after the 5th digit. Our two new children will be

11011 + 01000 and 00010 + 01101.

We can see that large chunks of each parent will survive to the next generation.

- ***More than one-point***

This is a generalization of 1-point crossover, in that we choose n places to splice up our solutions. Above, we could chose 3 points, after the 3rd, 6th, and 8th digit, to get

110 + 010 + 11 + 00 and 000 + 110 + 10 + 01

for the next generation.

- ***Uniform crossover***

Each new variable for the offspring is chosen randomly from each of the parent vectors. This works best when the variables are independent and therefore no relationship needs to survive to the next generation, only the values of the variables.

This works great for vectors, but not with permutations. We will almost always create illegal solutions by using these crossover techniques with say a path representation for the tour. More thought and time need to be devoted to gain intuition with different representations [10].

4.10 Some Application of Genetic Algorithms

The versions of the of the genetic algorithm described is very simple, but variations on the basic theme have been used in a large number of scientific and engineering problems and models. Some examples follow :

- ***Optimization:*** GAs have been used in a wide variety of optimization tasks, including numerical optimization and such combinatorial optimization problems as circuit layout and job-shop scheduling.
- ***Automatic Programming:*** GAs have been used to evolve computer programs for specific tasks, and to design other computational structures such as cellular automata and sorting networks.
- ***Machine Learning:*** GAs have been used for many machine learning applications, including classification and prediction tasks, such as the prediction of weather or protein structure. GAs have also been used to evolve aspects of particular machine learning systems,

such as weights for neural networks, rules for learning classifier systems or symbolic production systems, and sensors for robots.

- ***Economics:*** GAs have been used to model processes of innovation, the development of biding strategies, and the emergence of economic markets.
- ***Immune Systems:*** GAs have been used to model various aspects of natural immune systems, including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.
- ***Ecology:*** GAs have been used to model ecological phenomena such as biological arms races, host-parasite convolution, symbiosis, and resource flow.
- ***Population Genetics:*** GAs have been used to study questions in population genetics, such as "Under what conditions will a gene for recombination be evolution viable ?".
- ***Evolution and Learning:*** GAs have been used to study how individual learning and species evolution affect one another.
- ***Social Systems:*** GAs have been used to study evolutionary aspects of social systems, such as the evolution of social behavior in insect colonies, and, more generally, the evolution cooperation and communication in multi-agent systems [11].