CHAPTER THREE
GENETIC ALGORITHM FOR SOFTWARE TESTING AND RELATED WORK

3.1. Introduction

3.2. Basics of Genetic Algorithms

3.3. Parameters of Genetic Algorithms

3.4. Structure of Genetic Algorithms

3.5. How Genetic Algorithms work?

3.6. Objective Fitness Function and the Role of Feedback

3.7. Termination of the Genetic Search

3.8. Characteristics of Genetic Algorithms

3.9. GA for Software Testing

3.10. Literature Review of Related Work

3.11. Conclusion
Chapter 3: Genetic Algorithms for S/W Testing and Related Work

3.1. Introduction:

Optimization problems arise in almost every field, especially in the engineering world. As a consequence many different optimization techniques have been developed [110] including Simulated Annealing (SA) [115] and Genetic Algorithms (GA). Searching for an input datum in a set of possible input data that conforms to the test adequacy criteria, e.g. forcing traversing a specific path is a search problem.

The current literature identifies some specific types of search methods, direct search (Hill climbing), random search (random walks), etc [43]. Hill Climbing is a well known search algorithm. Hill climbing works to improve one solution, with an initial solution randomly chosen from the search space as a starting point, the neighborhood of this solution is investigated. If a better solution is found, then this replaces the current solution. The neighborhood of the new solution is then investigated. If a better solution is found, the current
solution is replaced again, and so on, until no improved neighbors can be
established for current solution. Hill climbing is simple and gives fast and sure
results. However, it is easy for the search to yield sub-optimal results when the
hill climbed leads to a solution that is locally optimal, but not global optimal.
Therefore, results obtained with hill climbing are highly dependent on the
starting solution [73].

Simulated annealing (SA) is similar in principle to Hill Climbing. However, by probabilistically accepting substandard solutions, simulated
annealing allows for less restricted movement around the search space. The
probability of acceptance \( p \) of an inferior solution changes as the search
progresses, and is calculated as:

\[
p = e^{-\frac{\delta}{t}}
\]

Where \( \delta \) is the difference in objective value between the current solution and
the neighbor inferior solution being considered, and \( t \) is a control parameter
known as the *temperature*, and the temperature is cooled according to a *cooling
schedule*. Initially, the temperature is high, in order to allow free movement
around search space, and so that dependency on the starting solution is lost. As
the search progress, temperature decreases. However, if cooling is too rapid, the
search space will not be enough explored, and the probability of the search
becoming stuck in local-optima is increased. The name “simulated annealing”
originates from the analogy of the technique with the chemical process of annealing – the cooling of a material in a heat bath. If a solid material is heated beyond its melting point, and then cooled back into a solid state, the structural properties of the cooled solid depend on the rate of cooling [73, 74].

Random search algorithms have achieved increasing popularity as the shortcomings of calculus-based schemes have been recognized. Random walk and random schemes search and save the best. Yet they are not-counted. Considering efficiency requirements, random searches in the long run can be expected to do no better than calculus-based search (hill climbing).

In general, these conventional search methods do not meet robustness requirements. Some are local in their scope, some are time-consuming and some lack efficiency when the space is too large to search [43].

Genetic algorithms are meta-heuristic search algorithms often used to solve complex optimization problems that are either very difficult or completely impractical to solve by other methods [75, 109].

The way defined in a biological sense, evolution takes place in organismic entities capable of encoding the structure of living organism. These are chromosomes. A similar process but with the same name – chromosome – is
used by genetic algorithms for encoding reasons [15]. Within an evolutionary frame of reference, change operates on the chromosomes and through them on the organisms which they decode. As in real life the simulated biological change occurs through mutation and crossover.

The central theme of research on genetic algorithms has been robustness, the balance between efficiency and effectiveness [73]. Thus where robust performance is desired, nature does it better. The secret of adaptation and survival are best learned from the careful study of biological examples. Yet genetic algorithms are not accepted method by appeal to this reason alone. Genetic algorithms are theoretically and empirically proven to provide robust search in complex spaces. Many papers and dissertations establish the validity of the technique in function optimization such as in software testing and automated test data generation. [73, 76, 105, 119 and 120].

Because genetic algorithms can search very large, highly non linear and often very noisy landscapes, they are ideal as solution engines for optimization problems. The reasons behind the growing number of applications are clear. These genetic algorithms are computationally simple yet powerful in their search for improvement.
A genetic algorithm is an enhancement of one of the earliest approaches in machine problem solving. The approach is known as generate and test (see Fig (3-1)).

Using this strategy, a new solution to the current problem state is generated and tested. If it satisfies the criteria for a solution, it is accepted; otherwise a new potential solution is generated and tested [21]. Because the generate-and-test method is always guided by an allowable outcome, it is called directed search.

Fig (3-1) The generate-and-test process.
Genetic algorithms are different from normal optimization and search procedures in four ways [43]:

1. GAs work with a coding of the parameter set, not the parameters themselves.
2. GAs search from a population of points not a single point.
3. GAs use payoff (objective function) information, not derivatives or other auxiliary knowledge.
4. GAs use probabilistic transition rules, not deterministic rules.

Taken together, these four differences contribute to genetic algorithms robustness and resulting advantage over other more commonly used techniques.

The genetic algorithm, as the name implies, breeds a solution to complex optimization and search problems using techniques that simulate the process of natural evolution. The concept underlying Genetic Algorithms is introduced in this chapter. The following sections explain the features, parameters and operators of Genetic Algorithms.

3.2. Basics of Genetic Algorithms

John Holland, the founder of Genetic Algorithms points out in [52] the ability of simple representations (bit string) to encode complex structures and the power of simple transformation to improve such structures. Holland showed that with the proper control structure, rapid improvements of bit strings could occur (under certain transformation). Population of bit string “evolves” as populations of animals do.
Chapter 3: Genetic Algorithms
for S/W Testing and Related Work

The process of breeding a genetic algorithms population over a series of
generations to arrive at the chromosome with the best fitness value is known as
convergence. The population converges to a solution.

An important formal result stressed by Holland is that even in large and
complicated search spaces, given certain conditions on the problem domain,
Genetic Algorithms would tend to converge on solutions that were globally
optimal or nearly so.

Convergence is controlled by the fitness function. The fitness function is
a measure associated with the objective function, which indicates the fitness of
a particular chromosome in terms of a desired solution. A genetic algorithm is
directed search. The search is controlled by the fitness function. For
minimization problems, the fitness function usually approaches zero as the
optimal value. For maximization problems, the fitness usually approaches some
upper boundary threshold as it’s optimal.

Genetic algorithms are search algorithms, learned from the careful study
of biological examples, in which a new set of individuals is created using bits
and pieces of the old individuals.
GAs has at least the following elements: population of individuals (chromosomes), selection method, crossover to produce new offspring and random mutation of new offspring [74]. Selection, crossover and mutation are known as GA operators. Crossover and mutation are responsible for reproduction, and these operators are discussed further.

### 3.2.1. Representation of Chromosomes (Coding):

In Genetic Algorithm, one possible solution to the problem is called individual, and all individuals together form a population. In Holland’s (and his followers’) work, individuals are bit strings consisting of 1s and 0s. The bit string is analogous to chromosomes in biological systems fig (3-2). Each chromosome contains a set of genes (bits in GA). Bit strings have been shown capable of representing usefully, variety of information, and they have shown as effective representations in unexpected domains.

The properties of bit string representations for genetic algorithms have been extensively studied. According to schema theorem, bit string coded implementations do have good performance.

In the proposed work binary representation for individuals is used, as it is natural to represent input data generated for testing objectives.
3.2.2. Selection Methods

Selection is a process through which high quality individuals (solutions) are chosen for subsequent generation of solutions. It concerns how to choose the individuals in the population that will create offspring for the next generation. The purpose of selection is to emphasize the fitter individuals in the population in hopes that their offspring will in turn have even higher fitness. This operator selects chromosome in the population for reproduction. The fitter the chromosome, the more it is likely to be selected to reproduce.
For reproduction, individuals are copied according to an objective function value $F$. We can think of function $F$ as some measure of profit, or goodness that we want to maximize. Copying individual strings according to their fitness values means that individuals with a high value have a higher probability of contributing one or more offspring in the next generation.

Numbers of selection methods have been proposed in the GA literature [43, 74]. The most used selection methods are described further.

3.2.2.1. Fitness-proportionate method:

In this selection method, the number of times an individual is expected to reproduce is proportional to its fitness. Individual fitness is divided by the average fitness of the population. The most common method is, “roulette wheel sampling”. Implementation of this method is described in Fig (3-3)

1. Calculate the fitness for each chromosome in the population.
2. Calculate the sum of all chromosomes fitness in population, (GS).
3. Generate random number $r$ from the interval $[0..1]$.
4. Loop through the population and sum the fitness $S$. until the sum $S$ is greater than $r$, stop and return the chromosome where you are,

*Step 1, 2 are performed only once for each population*

Fig (3-3) The process of roulette wheel selection method
3.2.2.2. Random Selection:

In this method, the selection made randomly so that every member of the current population has an equal chance of being selected for reproduction. A pseudo-code random number generator is used with uniform distribution to select the member of a generation to become parents for the crossover process. Pseudo-code for random generator is illustrated in Fig (3-4)

Mark selectable individuals of the current population from 1 to N.
For i=1 to pop-size do
    Generate random integer \( j \) from range \([1.. N]\).
    Select chromosome \( V_j \) from selectable individuals.
End for

Fig (3-4) Random selection method

3.2.3. Reproduction

Reproduction is a process in which individual strings are copied according to their objective values (fitness function). The GA process the selected parent of chromosomes to generate new chromosome (new population), in succession to replace one such population with another using crossover and mutation.
3.2.3.1. Crossover (recombination)

The process of “mating” [21] and reproduction in a genetic algorithm is called crossover.

Crossover is the Genetic algorithms’ power. It exchanges genetic material from two parents individuals (chromosomes), allowing useful inputs, i.e. good genes, from different parents to be combined in new chromosome (test case). Two individuals for crossover are chosen then crossover point $c$ is randomly selected between $I$ and $L$ where $L$ is chromosome length. Parent chromosomes are crossed at that point. The first child chromosome is identical to the first parent up to cross point $c$ and identical to the second parent after the cross point.

Although selection is surprisingly powerful, actually by itself it does not promote exploring new regions in search space. Therefore, no new points are found. If we are just copying old structures without changing them, how can we obtain something new? This is why crossover is used. The crossover roughly mimics biological recombination between two Single chromosome organisms [74].

Different crossover strategies are presented, and illustrated in Fig (3-5, 3-6, 3-7) respectively.
(a) **Single-point Crossover** is the simplest form. A single crossover position is chosen at random and the parts of two parents, after the crossover position are exchanged to form two offspring.

Fig (3-5) Single-point crossover at k=9.

<table>
<thead>
<tr>
<th>Parents</th>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100101010 0101010111 0000101100</td>
<td>1100101001 1111100000 1010101010</td>
</tr>
<tr>
<td>0011100001 1111100000 1010101001</td>
<td>0011100010 0101010111 0000101100</td>
</tr>
</tbody>
</table>

(b) **Double-point Crossover**: two positions are chosen at random and the segments between them are exchanged. Double-point crossover can combine more schemas than single-point crossover.

(c) **Uniform- Crossover**: is a multi-point crossover in which an exchange happens at each bit position. This operator takes two chromosomes and exchanges approximately half of the “genes” (bits). That is, at each bit position in parent A and parent B random
decision is made whither that bit should go into offspring A or offspring B.

<table>
<thead>
<tr>
<th>Parents</th>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100101010</td>
<td>0101010111</td>
</tr>
<tr>
<td>0011100001</td>
<td>1111100000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100100001</td>
</tr>
<tr>
<td>0011101010</td>
</tr>
</tbody>
</table>

Fig (3-6) Double-point crossover at k=5, m=13

<table>
<thead>
<tr>
<th>Parents</th>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100101010</td>
<td>0101010111</td>
</tr>
<tr>
<td>0011100001</td>
<td>1111100000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offspring</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001101011</td>
</tr>
<tr>
<td>0110101111</td>
</tr>
</tbody>
</table>

Fig (3-7) Multiple-point (uniform) crossover

There is no simple answer to “which crossover strategy one should use?” [74]. The success or failure of a particular crossover strategy depends on complicated way on the problem at hand, fitness function, encoding and other GA parameters.
3.2.3.2. Mutation:

One of the ways in which genetic algorithms attempts to improve the overall fitness of a population as it moves towards a final optimal solution is by randomly changing the value of a gene. This process is called mutation. This operation randomly flips some of the bits in chromosome. Mutation can occur at each bit position in a string with some probability.

Random mutation provides background variation and occasionally introduces beneficial material into the population. Without the mutation all individuals in population will sooner or later be the same, and there will be no progress any more. Program will be stuck in ‘local-maximum’.

3.3. Parameters of Genetic Algorithms:

In any GA implementation, we need to decide on a number of parameters: population size, crossover rate, mutation rate, and the number of generations. Often these values have to be “tuned” based on the results obtained. No general theory to deduce good values [66,112].

3.3.1: Population Size (pop-size)

Population is several numbers of individuals (candidate solutions). There is a large cost for evaluating a large population. Usually pop-size can range from
50-500 members. The advantage of using the population with many members is that many points in a space are searched in one generation.

In many genetic algorithms the population size remains the same from generation to generation. In others, the population size can expand or contract depending on the degree of genetic diversity, the rate of convergence, or other factors in the search process.

Different sizes of population works well with different problems and such sizes of population are used in the experiments of the proposed algorithm.

3.3.2: Crossover Rate ($P_c$):

Using this parameter, crossover occurs, or not, according to the probability of crossover $P_c$. It is simply the chance that two individual chromosomes will swap their bits. If $P_c=1$, crossover will always take place.

3.3.3. Mutation Rate ($P_m$):

After crossover, mutation flips each bit in a chromosome with the predetermined probability $P_m$. It is the chance that a bit within a chromosome will be flipped ($0$ becomes $1$, $1$ becomes $0$). This rate is usually very low value, say $0.001$[130]. Fig (3-8) illustrates the pseudo code for mutation.
Many mutation rates were suggested as optimum probability. Schaffer [104] and Muhlenbein et al [79] suggest that the probability $P_m$ is the reciprocal of the chromosome size so that it would be unlikely for the code to have an average more than one bit of a chromosome mutated. Others [21] proposed $P_m$ to be the reciprocal of population size, $P_m = \frac{1}{pop - size}$.

If $P_m$ is too low, there will be insufficient disruption for the population to prevent premature convergence to a ‘local-optimum’, and if the rate of mutation is significantly increased then it is a pure random search technique.

<table>
<thead>
<tr>
<th>Set mutation probability, $P_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>For i=1 to chromosome length do</td>
</tr>
<tr>
<td>{Create random number r,</td>
</tr>
<tr>
<td>If r &lt; $P_m$ then</td>
</tr>
<tr>
<td>Flip the i$^{th}$ bit in the chromosome,</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>End for</td>
</tr>
</tbody>
</table>

Fig (3-8) Pseudo code of random mutation
3.3.4. Number of Generations (*Max-gen*)

*Max-gen* is the maximum number of times we allow GA to iterate and generate test cases.

One of the primary termination conditions is a limitation on the maximum number of generations. *Max-gen* is dependent on program to be tested. And some test criteria are difficult to satisfy.

In the next chapters *max-gen* is ranged from 50-100 for different experiments.

It is worth mentioning that the stability, coherence, and convergence of genetic algorithms depend on the rate of mutation and the crossover frequency. Researchers have been trying to find the “best” GA parameter setting. However, optimal population size, crossover rate and mutation rate are likely to change over complexity of the problems search space [5,49,132 and 134].

3.4. Structure of Genetic Algorithms

The GA is a general method for solving “search for solution” problems. GAs have been used in a wide variety of tasks, including optimization automatic programming, evolution and learning, social and economics. The most common operations of a Genetic Algorithm include:
(a) **Selection for Reproduction:** this operation assigns the reproduction probability to each individual based on the output of the fitness function. The selection requires that the solution be evaluated for their fitness as parents: solution that is closer to an optimal solution is judged higher, or better *fit*, than others. After solutions have been evaluated, several solutions are selected in a manner that is biased towards the solutions with higher fitness values.

(b) **Crossover:** this operation is used to produce the descendants that make up the next generation. This operation involves the following procedures:

(i) select two individuals as a couple from the parent generation.

(ii) randomly select a position of the genes, corresponding to his couple, as a crossover point, thus each individual is divided into two parts.

(iii) exchange the first part of both genes corresponding to the couple.

(iv) add the two resulted individuals to the next generation.

(c) **Mutation:** this operation picks a ‘*gene*’ at random and changing its state according to the mutation probability. The purpose of the mutation operation is to maintain the diversity in a generation to prevent premature convergence to local optimal solution. The mutation probability is given intuitively, since there is no definite way to determine the mutation probability.
Upon completion of the above procedures, a fitness function should be devised to determine which of these parents and offspring can survive into the next generation. A block diagram of GA is shown in Fig (3-9).

These operations are iterated until the goal is achieved. Genetic algorithms guarantee high probability of improving the quality of individuals over several generations according to the Schema Theorem [43].
3.5 How Genetic Algorithms work?

This fundamental question has been not answered for long time. Much work has been done on the theoretical foundation of GAs (e.g. [43] and [52]). The first traditional theory formulated by Holland assumes that, at every general level of description, GA work by discovering, emphasizing and recombining good “building blocks” of solutions in highly parallel fashion. The idea here is that good solutions tend to be made up of good building blocks.

Holland introduced the notion of schemas (or schemata) to formalize the informal notion of “building blocks”. Schema theorem [43, 52 and 74] describes the growth of a schema from one generation to the next.

Crossover is believed to be the major source of GA’s power, with the ability to recombine instances of good schema to form instances of equally good or better schema [74]

Combination of selection and crossover increases the number of schema. Holland showed that the number of schema, which are effectively being processed in each generation is of the order $n^3$, where $n$ is the population size. This property is the explanation of the good performance of GA.
Spears and Anand [107] demonstrated that using genetic algorithm with mutation but without the crossover achieves a respectable result for small population where diversity is limited.

Experimental results by Dejong (1990) and Schaffer and Eshelman (1991) cited in [110] confirmed that mutation is a powerful operator for GA, but without crossover the GA is not always sufficient.

Neither mutation nor crossover alone should be advocated or dismissed. Each operator plays an important and different role in search process, where crossover is better for achieving good solution because crossover shares information between fit individuals, whereas mutation is better for disruption and exploration.

Genetic Algorithms are suitable for solving a test data generation problems. Each test data will be coded in bit string and represent individual and the test suite will be the population.

**3.6. Objective Fitness Function and the Role of Feedback:**

A collection of chromosomes with their values is a population. As the algorithm continues, new populations of old and new chromosomes are created and processed
In GA when a good solution has been generated, the effectiveness of GA in optimizing parameters also depends on the usefulness of the information obtained via feedback. It is important that the right feedback mechanism is selected to enable an adaptive search strategy. Feedback mechanism is called fitness, cost or payoff functions.

The genetic algorithms are blind; they know nothing of the problem except the fitness information. A good fitness function will provide useful information early in the search process to help focus attention throughout search space, and it is considered in the proposed work.

When the population is largely converged, competition among population members is less strong and the algorithm tends to wonder. In this case objective fitness function value must be scaled up to bring out differences between population members to continue to reward the best performers [74]. Scaling of objective fitness values has become a widely accepted practice. The literature proposed many scaling methods [74]. These insure that average population members receive one offspring copy on average and the best receive the specified multiple number of copies. In the late runs of the genetic algorithm, there may still be significant diversity within the population; however, the population average fitness may be close to the population best fitness. If this situation is left alone, average members and best members get
nearly the same number of copies in future generations, and the survival of the fittest necessary for improvement become a random walk among the mediocre. In this case **fitness scaling** can help in convergence.

One useful scaling procedure is linear scaling Fig (3-10). With the raw fitness $f$ and the scaled fitness $f'$, Linear scaling requires a linear relationship between $f'$ and $f$ as follows:

$$f = af + b$$

The coefficient $a$ and $b$ may be chosen in a number of ways. Other types of scaling procedure can be found in [43, 52 and 74]

![Linear Scaling](image)

*Fig (3-10) Linear Scaling under normal conditions.*
3.7. Termination of the Genetic Search

A genetic algorithm must be informed when to stop searching for a solution. Each new population is known as a generation. A maximum number of generations is one of the termination conditions for a genetic algorithm.

There is several common termination conditions used in genetic algorithms:

1. The maximum number of generations have been reached.
2. A maximum elapse time has been reached.
3. A maximum amount of computer resources have been used.
4. The average fitness function value reaches a steady state over successive generations.
5. The average fitness reaches a very early steady state or begins to decay.

3.8. Characteristics of Genetic Algorithms

GA is better than the conventional search methods in that it is more robust, unlike older systems they do not break easily even if the inputs change slightly [87].

Strength and limitations of genetic algorithms can be as follows [22]:

- The ability to solve nonlinear, noisy, and discontinuous problems.
- The ability to solve complex optimization problems.
• A complete dependence on the fitness function: as directed search technique, the fitness function evaluates each solution and ranks it according to a goodness of fit. If a fitness function cannot be defined, a genetic algorithm cannot be used to solve the problem.

3.9. GA for Software Testing:

Genetic algorithms belong to a class of directed search methods that are used for both solving optimization problems and modeling the core of evolutionary systems.

In addition to the goal of finding better and better chromosomes, breeding has the goal of increasing diversity in the population. Only through genetic diversity can a genetic algorithm economically and effectively explore a sufficient portion of the solution space.

Translating the concept of genetic algorithm into a working engine evolves not only designing way to represent the basic data structure but also ways of setting the principal parameters of the genetic algorithms.

Software testing using an automatic test program will generally avoid the errors that humans make when they get tired after multiple repetitions [4].
In the context of software testing, the basic idea is to search the domain for input variables which satisfy the goal of testing (test criteria). The test aim must itself be transformed into an optimization task. For this, a numeric representation of the test aim is necessary, from which a suitable fitness for the evolution of the generated test data can be derived.

GA searches from a population of points rather than a single point. The initial population compromises a set of individuals generated randomly or heuristically. The members that are fitter are given a higher probability of participating during selection and reproduction phase, and the others are more likely to be discarded. Fitness is measured by decoding a chromosome into the corresponding variables on an objective function. The value returned by the objective function is used to calculate the fitness value. Test cases evolve from one generation to the next to generate optimal test suite that satisfy the test criteria.

GAs is quite suitable in solving problems of many types, which have to be adjusted towards a global optimum. The bit string of the variables are then concatenated together to produce a single string (chromosome) which represents the whole vector of the variables in the problem.
Test case generation by GA can be compared to another common automatic technique, random generation.

GAs form a method of adaptive search in the sense that they modify the test data from one generation to the next, in order to optimize a fitness function. In contrast, random testing generates test data uniformly without any knowledge of previous test data [55]. However, as [28] points out that random testing is inexpensive as all that is required is a random number generator, and it is more stressing the program under test than manual generation.

Research by [117] and [2] also concluded that random testing was a useful validation tool. The down-side to random generation is the lack of full coverage of a SUT.

In the next section we present a literature review of the related work done on automatic test data generation using Genetic Algorithms.
3. 10. Literature Review of Related Work:

The survey paper presented by Sandler [103] addresses the various testing tools available. The survey covers a few commercial off-the-shelf tools such as GUI tester named GA Wizard, test case generators named JTest and Cantata++ and a fault injection program called Holodeck.

The survey focus on test case generators and the use of genetic algorithms for path testing, specially the work done by [66] (2001). Sandler did not find the formula for the weight presented by [66] meaningful because they didn’t state the applicability of that formula to other problems than triangle-classifier problem.

Alander et al [1] presented a method to produce test case in order to find problematic situations like processing time extreme. The proposed test case generator comes under the heading of automated dynamic stress testing. They try to identify the situation where the software has the lowest reaction time determined with difficult inputs generated by GA using black box technique.

GA sends input to the black-box tested program and measures response time. The response time is the fitness value for the GA.
They noted that the response time is somewhat non-deterministic when the same inputs are given, different response times are obtained. These can interfere with the search made by GA.

**Alander et al** [2] presented a study of testing real times programs by genetic algorithms. The authors proposed that GA could be used to produce test cases to find problematic cases e.g. the maximum response time. The authors applied black-box testing on electronic network protection relay software. The authors claim that yet no results were obtained because of difficulty and time constraints.

**Berndt et al** [8] explored the use of GA to test programs of a distributed system. No structural analysis of the code is performed. The aim of their work is functional testing or black-box that seeks to confirm that functions correctly implemented specifications.

The fitness function depends on results from previous testing cycles. Relative rather than absolute fitness function based on the concept of novelty, proximity and severity. Keeping record of the past generations in a fossil record, allowing any current fitness calculation to be influenced by past generations.
Another work by Watkins et al [117] reports in using GA for test case generation for system level testing. They explained that it is very difficult for human to test and analyze the failure patterns that are embedded in large, multidimensional spaces of successful executions. And that the GA data is a rich data source that can be used to assist testers in pinpointing errors and demonstrate the effectiveness of the genetic algorithms. They used fitness function which was relative rather than absolute for the generation of test cases. They keep a fossil record to store changes of fitness function. Then they reward the individual based on the concept of novelty, approximity and severity. They have listed the attributes as a parameter for system testing prospective.

A fitness of test case is based on its proximity to errors and other test cases, its novelty and whether it causes an error.

The limitation of their research work [117] is that the errors were all expressed in terms of system attributes.

In the work done by Berndt et al [7] the authors focus on automatic test data generation on high volume testing, using genetic algorithms. The authors described example on complex distributed systems and failures such as: Patriot missile systems, London Ambulance Service, German Railway Switching, Advance Telephone System and NASA Mars Rovers.
In their work, fitness function calculation depends on results from previous testing cycles. That is, a relative than absolute fitness function is used. The authors experimented with autonomous vehicle simulation that involves simple robots that explores a mock landscape, collecting samples and transmitting data. The strategy used is black-box testing and regression testing.

**Alireza Rezaee** [3] employed GA to generate set of test cases for constraint automata's Black-box testing of XML components, which is a modeling language for XML. Each element of XML such as a black-box component must be mapped to constraint automata.

The test data was generated for the coverage of transactions and states, on wide range of scenarios. His fitness function is defined as:

\[
\text{fitness}_{GA} = \frac{(n + m)^5}{L}
\]

Where \( n \) and \( m \) are the number of different transitions and states respectively, and \( L \) is the length of the test case.

Another approach used was the GSA genetic Symbiosis Algorithm. Where, fitness of each individual is dependent to other individuals. The author concluded that the proposed algorithms obtained the coverage of (100%) after 500 generations.
While the above work considers black-box testing, other research work was done to consider white-box testing.

**Korel** [62] is the first to present dynamic approach to test data generation, which is based on actual execution of a program under test, dynamic data flow analysis and function minimization methods. Test data are developed using the actual values of input variables and his work shows that the problem can be reduced to a sequence of sub-goals where each sub-goal is solved using function minimization search techniques.

Branch coverage criterion is considered in his work using direct search method which progress towards the minimum using a strategy based on the comparison of branch function values only.

He assumed that all branch predicates are of the form E1 op E2, (where E1, and E2 are arithmetic expressions and op is one of {<, >, <=, >=, =, !=}) furthermore, it is assumed that the predicates do not contain ANDs or ORs or other Boolean operators. Each branch predicate can be transformed into the equivalent predicate.

**Watkins** [118] has applied GA to the testing problem using different approach in applying fitness function. Watkins uses fitness function based on
the number of times a particular path is executed. This means that the more often the same path is taken then the lower will be the fitness value.

Michael et al [76] implemented Korel’s [62] function minimization approach in their GA based test data generation. Their generator has the condition-decision coverage as its adequacy criteria. They used the test data generation algorithm on two moderately sized programs as well as small programs that have frequently been used as benchmarks for test data generation techniques. They have reported preliminary results from an experiment comparing random test data with their GA approach. Their results show that GA search outperformed random test data generation.

Borgelt [11] investigated how a genetic algorithm may be used to produce software unit test data for use in structural testing. His work studied the effect of varying the population size as a function of module complexity. His fitness function based on the probability of traversing the last block of code in the path through the module, and a “bonus” to help GA search for hard to reach code fragments.

The author experimented with triangle classifier program with variable population sizes and 24-bit chromosome length and 200 runs.
His results were compared with Watkins [118] algorithm results to investigate the performance of fitness function of both algorithms.

**Pargas et al** [90] presented test data generator that uses GA to generate test data for statement and branch coverage. Their experiment involved small set of C programs. The fitness value of test cases was computed as follows: if test case \( i \) contains more predicates in control flow graph (CFG) for target path than test case \( j \), then assign test case \( i \) a higher fitness value than test case \( j \). Then the algorithm sorts the population based on their fitness value. They implemented their technique with a tool-called TGen in which parallel processing was used.

The authors explained their way to execute the TGen to provide path and def-use coverage but no experiments were shown.

**J. Wittakar** [123] provided a close look at software testing within the context of software development. The author presented a sample software testing problem to show some of the testing difficulties.

The author gives important points to testers that must be taken into consideration such as file format, communication protocols, while going through the different phases of the testing process (i.e. modeling software’s
environment, selecting test scenarios, running and evaluating test cases and measuring test progress). The author finally provides helpful test adequacy criteria for structural and functional testing.

The work done by Sthamer et al [109] included the use of genetic algorithms in the generation of test cases for branch coverage for temporal behavior with regards to its exceeding or falling below a specified timing constraints and also testing safety properties of embedded systems. The conducted case studies of industrial applications concentrating on the assessment of testability of programs. The results of their work showed that in the case of temporal behavior test found the longest execution time for all the programs. And that result is better than random testing.

While the work on testing safety properties of the embedded systems is still in infancy as they claimed.

Lin & Yeh [66] presented a genetic algorithm to automatically generate test cases to test a selected path. Metric named normalized extended humming distance (NEHD) is developed. Based on NEHD, a fitness function named SIMILARITY is found. They claimed that quality of test cases generated by SIMILARITY for path testing is higher than the quality test case produced by
random way. They experimented with triangle-classifier program and, their test data generated were of integer type only.

_**Wegener et al** [121] presented test generator for structural method specifically statement and branch coverage of real world embedded software. The proposed fitness function consists of two major building blocks: approximation level and normalized predicate local distance. Over all fitness value is the summation of the approximation level and the local distance value. An individual with fitness value (zero) means that it satisfies the partial aim. The stopping criteria used are full coverage and number of generations, depends on which one is satisfied first.

It is reported that full coverage of some programs achieved but not for all. They compared results with test data that are generated using random testing, which apparently result in much lower coverage.

_Sami et al_ [101] proposed a path oriented test case generation. In their approach, test cases determined using binary search, which requires certain assumptions. The path to be covered is not considered as a whole. It is rather divided into its basic constituent (i.e. edges) and the input need to fulfill the branching condition of each edge for its traversal.
Sami et al claimed that the results of conducted case study showed that the algorithm can require far less iterations than other comparable approaches. But no experimental results of any empirical study were shown or discussed to verify their claim.

Mansour & Salame [69] stated that there are 5 levels of software testing (unit, integration, product, system, and acceptance) testing. They also explained (White-box and Black-box) testing. The work concerns with unit testing.

They formulate a path testing problem as an optimization problem, where objective function is based on control flow graph of a program and derived from the program predicates that define the path. They presented a simulated annealing (SA) algorithm to determine test cases that traverse a selected path to be covered. They also proposed Genetic algorithm (GA) that generate input data for both integer and real value data, and then compared SA with GA.

In their GA work, Path predicates are expressed in terms of constants and input variables. The objective function is given by weighted hamming distance between the operands of each predicate which is a count base of the number of different bits of the two operands. The objective function for single predicate is given by: $Z_i=\Sigma 2^i$
Where the sum is done over all indices $i$, where two bits differ. The objective function for each predicate is determined separately then overall value is calculated by using multiplication for the OR-ed predicated and addition for the AND-ed predicates. The authors experimented with 8 subject programs.

The approach has limitations. It works only for numerically valued predicates, also not directly applicable to predicated that involves strings, Boolean and array elements.

The authors concluded that SA performed slightly better than GA however GA is faster than SA.

In the work done by M. Girgis [40], the author presented an automatic test data generation technique that uses a genetic algorithm (GA) which is guided by the data flow dependencies in the program to search for test data to cover its def-use associations.

His fitness function was: \[ \text{fitness} = \frac{\text{no of goals covered by case}}{\text{total no of goals need to be covered}} \]

In the GA’s parent selection process, the author uses one of two methods: the roulette wheel method, and a proposed random selection method. The author evaluates the effectiveness of his GA algorithm by comparing the
experimented results with the random testing technique. And he compared his proposed random selection method, to the roulette wheel method, which he claimed to give better results.

**de Abreu et al** [23] proposed the Generalized Extremal Optimization (GEO) technique for the problem of automatic test data generation. They claimed that GEO is competitive to Simple GA in their initial assessment, since the tuning process for GEO is more simple than SGA. They experimented with one program, triangle-classifier and listed results comparing GEO to SGA and Random testing. No loop testing or data inputs other than integer numbers type were considered.

### 3.11. Conclusion

The conventional search methods do not meet robustness requirements [43]. Genetic algorithms are used to solve complex optimization problems. GAs are computationally simple, yet powerful in their search for improvement. The concept of Genetic Algorithms, their features, parameters and operators is introduced. The objective fitness function and the role of feedback are also explained.

GA is a quite suitable in solving automatic test data generation problem to avoid errors that human makes. A random test data generator may create
many test data, however, because information about the test requirement are not incorporated into the generation process, the test data generator may fail to find test data that satisfy the requirement of testing.

The literature review in this Chapter has covered different varieties of optimization techniques including simulated Annealing and genetic algorithms [69]. The GA-based test data generators were applied for white box testing as well as black-box testing. The researchers work done on automatic test data generation using GA is thoroughly surveyed. Different fitness functions were developed by many researchers for different coverage criteria such as statement coverage [90, 121], Branch coverage [62, 109], condition/decision coverage [76], def-use coverage [40] and path coverage [66, 101, 133].

The proposed algorithm to enhance the automatic generation of test data for path testing is presented further.