Chapter I
Introduction

1.1. Introduction

Software systems have become an integral part of all engineering systems and are providing assistance in a variety of engineering activities. However, the quality of assistance provided depends not only on the coded functionality, but also on the fact that the software is free from faults. This makes testing of software systems an extremely important activity. However, as both research and practice have repeatedly pointed out, there exist numerous technical and conceptual difficulties in its implementation. Beginning with the derivation of a feasible, cost efficient and adequate set of test data, the ability to execute a program on selected test data, deciding whether the outcome of the test is valid, deciding whether one has tested sufficiently and can stop presents many challenges. More importantly, with the increasing size and complexity of software systems, there is also an urgent requirement for automating the process of software testing and, in particular, address the issue of automated test data generation (McMinn, 2004; Bertolino 2007; Harman, 2007). With the realization that the process of software test data generation can be cast into a search problem and that metaheuristic techniques can be readily applied, a new area of research called search-based software test data generation has emerged (McMinn, 2004). However, as research points out, the application of metaheuristic techniques poses new challenges. Keeping in mind the challenges outlined in the published literature, the following have been addressed in this thesis in the context of test data generation for branch coverage using population based metaheuristic techniques:

1. The problem of ensuring that the current population contains an individual that encodes inputs for which the execution path reaches the predicate node of the target branch. The idea of branch ordering for target selection together with memory and elitism is explored in this context.

2. Exploiting and comparing different metaheuristic techniques.

3. A combinational approach to fitness function design.
A structured, master-slave cum island based, parallel approach to test data generation using the extended path prefix strategy.

Fitness function design using a maximization approach.

Further in this chapter, we describe test data generation and its automation and metaheuristic techniques.

1.2. Test Data Generation and its Automation

Test data generation in software testing is the process of identifying a set of test data to satisfy a selected test criterion (Korel, 1990a). Although test data can be generated manually, an automated solution to the test data generation problem attempts to automatically create a test set $T$ that will satisfy selected test data adequacy criterion $C$ when it is used to test program $P$. (Bertolino, 2007) points out that the most promising results towards automated test data generation have come from three approaches: model-based, random and search-based approaches.

According to Utting (Utting, 2005), model-based testing is the automation of specification-based testing. Utting et al. (Utting et al., 2006) further define model-based testing as the automatable derivation of test data from abstract formal models and their execution. Models must be formal enough to allow, in principle, a machine to derive tests from these models. However, the manual derivation of test data from formal models is also in the scope of model-based testing. Models may be described using different notations, for instance using pre/post conditions as in JML (Leavens and Cheon, 2006) and finite state machines as in UML state charts (Fowler and Scott, 1999). Depending on the chosen test criterion, which is usually based on the model, test data may be generated using various techniques such as random generation, graph search techniques and symbolic execution (Utting, 2005; Utting et al., 2006).

Random test data generation simply consists of generating test data at random until a useful test data is found. In terms of achieving a specified degree of adequacy, this approach does not perform well as compared to systematic approaches and has been traditionally used for benchmarking. However, recent implementations of feedback-based approaches to random testing and combining random testing with symbolic execution have shown improved performance (Bertolino, 2007).
Search-based test data generation is a part of the much broader research area of \textit{search-based software engineering} (Harman, 2007). Search-based test data generation consists of exploring the input domain $D$ of the program under test $P$ for test data to satisfy a selected test criterion $C$. By using metaheuristic techniques, search is directed towards the most promising areas of $D$ (Harman 2007; McMinn, 2004; Michael et al., 2001). As a technique, it can be used with specification-based criteria as well as program-based criteria (Bertolino, 2007).

\subsection*{1.2.1. Dynamic Test Data Generation}

Search-based software test data generation has its roots in the program-based, \textit{dynamic test data generation} methodology proposed by Korel (Korel, 1990a; Korel, 1992; Ferguson and Korel, 1996). In this section we first review program-based, dynamic test data generation and then introduce search-based test data generation in its context.

A \textit{control flowgraph} (or \textit{flowgraph}) is a graphical representation of a program under test. A flowgraph of a program $P$ is a directed graph $G = (N, E, s, e)$ consisting of a set of nodes (or vertices) and a set of edges $E \in N \times N$. Each node represents a basic block which is a linear sequence of instructions, where the flow of control enters in the beginning and leaves at the end without halt or possibility of branching except at the end. Each edge $(n_1, n_2)$ represents a transfer of control from $n_1$ to $n_2$ and is associated with a branch predicate that represents the condition of transfer. The branch predicate might be the empty predicate which is always true. If a node has more than one outgoing edge we refer to the node as a \textit{predicate node} and the edges as \textit{branches}. In each flowgraph there are two special nodes, an \textit{entry} node $s$ and an \textit{exit} node $e$. The start node has no inward edges and the exit node has no outward edges. Figure 1.1b shows the control flowgraph for the program in Figure 1.1a.
```c
int triangle_type(int a, int b, int c) {
    int type = PLAIN;
    if (a > b) { swap (a, b); }
    if (a > c) { swap (a, c); }
    if (b > c) { swap (b, c); }
    if (a == b) {
        if (b == c) {
            type = EQUILATERAL;
        } else {
            type = ISOSCELES;
        }
    } else if (b == c) {
        type = ISOSCELES;
    }
    return type;
}
```

Figure 1.1 A program and its control flowgraph

A path is a sequence of nodes $p = (n_1, n_2, ..., n_q)$ where $n_q$ is the last node of the path $p$ and $(n_i, n_{i+1}) \in E$ for $1 \leq i < q$. Whenever the execution of $P$ on $d \in D$ traverses a path $p$, we say that $d$ traverses $p$. In order for execution to continue through a branch the corresponding
branch predicate has to be true. Thus the set of all branch predicates in a path is termed as *path predicate*. A path is *feasible* if there exists an input $d$ in domain $D$ of $P$ that traverses the path, otherwise the path is *infeasible*. A path that begins with the entry node and ends with the exit node is called a *program path*. Otherwise, it is sometimes called a *path segment*.

The process of dynamic test data generation is illustrated in Figure 1.2 (Kapfhammer, 2004). During dynamic test generation, the source code of a program is instrumented to collect information about the program as it executes. The resulting information, collected during each test execution of the program, is used to heuristically determine how close the test data is to satisfying a specified test requirement as specified by the selected test criterion. This allows the test generator to modify the program's inputs gradually, moving them ever closer to values that actually do satisfy the requirement. In other words, the problem of generating test data reduces to the well understood problem of function minimization. Furthermore, dynamic test generation methods can handle arrays and pointer references because the values of array indices and pointers are known throughout the generation process.

For example, consider the condition $(a>c)$ on line 3 of the program in Figure 1.1a. If the test requirement is to ensure that the branch predicate $(a>c)$ is TRUE, then a test data must be found to cause the value of $c$ to be less than $a$ when line 4 is reached. The program can be executed and the value of $a$ and $c$ can be recorded. If $a_x$ and $c_x$ denote the value of $a$ and $c$ for test data $x$ (in this example, a triple) at line 4, then the function,

$$f(x) = \begin{cases} c_x - a_x, & \text{if } c_x < a_x \\ 0 & \text{otherwise} \end{cases}$$

is minimal when the TRUE branch is taken. Thus the problem of test data generation is reduced to function minimization. Although the test requirement has been illustrated here as a branch predicate, it can be a path or a complete path or any other requirement depending on the selected test criterion (McMinn, 2004). Furthermore, the test criterion itself need not be program-based, it can be specification based as well (McMinn, 2004).

More importantly, the idea of function minimization opens up the possibility of application of a wide range of optimization techniques ranging from the classic to the metaheuristic (Harman, 2007; McMinn, 2004).
According to McMinn (McMinn, 2004), “Metaheuristic search techniques are high-level frameworks which utilise heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. Such a problem may have been classified as NP-complete or NP-hard, or be a problem for which a polynomial time algorithm is known to exist but is not practical.” Search techniques such as simulated annealing, evolutionary algorithms (Michalewicz and Fogel, 2000) and other new ideas in optimization such as particle swarm optimization (Eberhart et al., 2001) are metaheuristic in nature. Given that problems in software engineering, in general, are often multi-criteria and augmented versions of NP complete problems (Harman, 2007), they are suited to the application of metaheuristic techniques.

To the problem of test data generation, metaheuristic techniques such as simulated annealing (Tracey et al., 1998a), evolutionary algorithms (McMinn, 2004; McMinn and Holcombe, 2006, Michael et al., 2001) have been widely used. Recently, techniques such as particle swarm optimization (Windisch et al., 2007), quantum particle swarm optimization (Agarwal and Srivastava, 2010) and artificial immune systems have also been used.

1.3. Evolutionary Algorithms

Evolutionary Algorithms (EAs) represent a class of metaheuristic techniques inspired by nature (Michalewicz, 1996; Alba and Tomassini, 2002; Michalewicz and Fogel, 2000). Population based evolutionary algorithms include (Goldberg, 1989; Michalewicz, 1996; Streichert, 2002; Michalewicz and Fogel, 2000) genetic algorithms (GA), evolutionary strategies (ES) (Rechenberg, 1994), evolutionary programming (EP), genetic programming (GP), particle swarm optimization (PSO) (Kennedy and R. Eberhart, 2001), and estimation of distribution algorithms (EDAs). In these algorithms, a population of candidate solutions competes for survival from one generation to the next and as new solutions are parented, the population gradually evolves to contain the optimal or near optimal solution. Individual members of the population represent candidate solutions and are typically represented as strings of binary or real-valued parameters, although other representations such trees, linked lists, etc. have also been used (Michalewicz, 1996; Michalewicz and Fogel, 2000).

The steps in a standard EA are outlined as follows (Alba and Tomassini, 2002):
t = 0;
Initialize(P(t));
Evaluate(P(t));
while (not stop-condition) {
    P'(t) = Variation(P(t));
    Evaluate(P'(t));
    P(t+1) = Select(P'(t), P(t));
    t = t+1;
}

An EA proceeds in an iterative manner by generating new populations of candidate solutions (individuals) $P(t)$ from the old ones ($t = 0, t = 1, \ldots$). Beginning with an random initial population $P(0)$, an evaluation function, Evaluate(), associates a fitness value with each individual indicating its optimality with respect to the problem. A whole new intermediate population of individuals, $P'(t)$ is produced by applying the function Variation() to $P(t)$. This function may involve a number of steps in which different operators are applied to individuals or group of individuals. For instance, in the standard GA (Goldberg, 1989) the stochastic operators of selection, crossover and mutation are applied. $P'(t)$ is evaluated using Evaluate() and the next generation population $P(t+1)$ is formed by another process of selection in function Select().

In this thesis we explore the application of the following algorithms:

- Genetic Algorithm
- Binary Particle Swarm Optimization
- Quantum Particle Swarm Optimization
- Clonal Selection Algorithm

A detailed description of these algorithms is presented in the next few sections.

### 1.3.1. Genetic Algorithm

Genetic Algorithm (GA) is a search algorithm that is based on the idea of genetics and evolution in which new and fitter set of string individuals are created by combining portions of fittest string individuals of the parent population (Goldberg, 1989). A genetic algorithm execution begins with a random *initial population* of candidate solutions $\{s_i\}$ to an objective function $f(s)$. Each candidate $s_i$ is generally a vector of parameters to the function $f(s)$ and usually appears as an encoded binary string (or *bit string*) called a *chromosome* or an
individual. An encoded parameter is referred to as a gene, where the parameter’s values are the gene’s alleles. If there is \( m \) input parameters with the \( i \)th parameter expressed in \( n_i \) bits, then the length of the chromosome is simply \( \sum_{i} n_i \). In this thesis each individual, or chromosome, represents an encoding of test data.

After creating the initial population, each chromosome is evaluated and assigned a fitness value. Evaluation is based on a fitness function that is problem dependent. From this initial selection, the population of individuals iteratively evolves to one in which candidates satisfy some termination criteria or, as in our case, fail to make any forward progress. Each iteration step is also called a generation.

Each generation may be viewed as a two stage process (Goldberg, 1989). Beginning with the current population (also called the parent population), selection is applied to create an intermediate population and then recombination and mutation are applied to this population. Another (selection) step is then applied to the individuals from the intermediate and the current generation parent population to create the parent population for the next generation. In generational GA, the intermediate population replaces the current generation’s parent population to create the parent population for the next generation whereas in the steady state GA a small percentage of worst individuals from the parent population is replaced with best individuals from the intermediate population. In the case of generational GA, elitism ensures that the fittest chromosomes survive from one population to the next.

Crossover, i.e., recombination, is applied to randomly paired individuals with a probability. Amongst the various crossover schemes are the one point, two point and the uniform crossover schemes (Goldberg, 1989). In the one point case a crossover point is identified in the chromosome bit string at random and the portions of chromosomes following the crossover point, in the paired chromosomes, are interchanged. In addition to crossover, mutation is used to prevent permanent loss of any particular bit or allele. Mutation application also introduces genetic diversity. Mutation results in the flipping of bits in a chromosome according to a mutation probability which is generally kept very low.

The chromosome length, population size, and the various probability values in a GA application are referred to as the GA parameters in this thesis. Selection, crossover, mutation are also referred to as the GA operators.

1.3.2. Particle Swarm Optimization and Binary Particle Swarm Optimization
Particle Swarm Optimization (PSO) was initially proposed to find optimal solutions for continuous space problems by Kennedy and Eberhart in 1995 (Kennedy and Eberhart, 1995). In PSO the search starts with a randomly generated population of solutions called the swarm of particles in $d$-dimensional solution space. Particle $i$ is represented as $X_i = (x_{i1}, x_{i2}, \ldots, x_{id})$ which is called the position of the particle $i$ in $d$-dimensional space. With every particle $i$ a velocity vector $V_i = (v_{i1}, v_{i2}, \ldots, v_{id})$ is associated that plays an important role in deciding next position of the particle and is updated in each iteration. For updating the velocity of each particle, the particle’s best $P_{ibest}=(p_{i1},p_{i2},\ldots,p_{id})$ which is the best position of particle $i$ achieved so far and global best $P_{gbest}=(p_{g1},p_{g2},\ldots,p_{gd})$ which is the best position of the swarm achieved so far by any particle of the swarm, are used. Following equations (1.3) and (1.4) are used to find new velocity and position of particle $i$ in iteration $t+1$.

$$V_i(t+1)=w.V_i(t)+c_1\phi_1(p_{ibest}-X_i(t))+c_2\phi_2(p_{gbest}-X_i(t)) \quad (1.2)$$

$$X_i(t+1)=X_i(t)+V_i(t+1) \quad (1.3)$$

In equation (1.2), $w$ is the inertia weight which controls the impact of previous history of velocity on global and local search abilities of particles (Dorigo and Stützle, 1999), $c_1$ and $c_2$ are positive learning constants which determine the rate by which the particle moves towards individual’s best position and the global best position. Usually, $c_1$ and $c_2$ are chosen in a way so that there sum doesn’t exceed 4. If it exceeds 4 at any time then both the velocities and positions will explode toward infinity. $\phi_1$ and $\phi_2$ are random numbers drawn from uniform probability distribution of $(0,1)$. In this way positions and velocities of the particles are evolved in each iteration until the optimal solution is not obtained.

In 1997 Kennedy and Eberhart (Kennedy and Eberhart, 1997) introduced the binary particle swarm optimization (BPSO) algorithm. In the binary version every particle is represented by a bit string and each bit is associated with a velocity, which is the probability of changing the bit to 1. Particles are updated bit by bit and velocity must be restricted within the range $[0,1]$. Let $P$ be the probability of changing a bit from 0 to 1, then $1-P$ will be the probability of not changing the bit to 1. This probability can be represented as the following function:

$$P(x_{id}(t)=1) = f(x_{id}(t),v_{id}(t-1),p_{sd},p_{gd}) \quad (1.4)$$

where $P(x_{id}=1)$ is the probability that an individual particle $i$ will choose 1 for the bit at the $d^{th}$ site in the bit string, $x_{id}(t)$ is the current state of particle $i$ at bit $d$, $v_{id}(t-1)$ is a measure of
the string’s current probability to choose a 1, $p_{id}$ is the best state found so far for bit $d$ of individual $i$, i.e., a 1 or a 0, $p_{gd}$ is 1 or 0 depending on what the value of bit $d$ in the global best particle.

The most commonly used measure for $f$ is the sigmoid function which is defined as follows:

$$f(v_{id}(t)) = \frac{1}{1 + e^{-v_{id}(t)}}$$

(1.5)

where,

$$v_{id}(t) = w v_{id}(t-1) + (\phi_1)(p_{id} - x_{id}(t-1)) + (\phi_2)(p_{gd} - x_{id}(t-1))$$

(1.6)

Equation (1.6) gives the update rule for the velocity of each bit, where $\phi_1$ and $\phi_2$ are random numbers drawn from the uniform distributions. Sometimes these parameters are chosen from the uniform distribution, such that their sum is 4. The $v$ value is sometimes limited so that $f$ does not approach 0.0 or 1.0 too closely. In this case, constant parameters [$V_{min}$, $V_{max}$] are used. When $v_{id}$ is greater than $V_{max}$, it is set to $V_{max}$ and if $v_{id}$ is smaller to than $V_{min}$, then $V_{id}$ is set to $V_{min}$. This simply limits the ultimate probability that bit $x_{id}$ will take on a zero or one value. A higher value of $V_{max}$ makes new vectors less likely. Thus $V_{max}$ in the discrete particle swarm plays the role of limiting exploration after the population has converged (Kennedy and Eberhart, 1997), i.e., it can be said that $V_{max}$ controls the ultimate mutation rate or temperature of the bit vector. Smaller $V_{max}$ leads to a higher mutation rate (Kennedy and Eberhart, 1997).

### 1.3.3. Quantum Particle Swarm Optimization

More recently the potential advantages of parallelism offered by quantum computing (Yang et al., 2004) through superposition of states in qubit registers, and simultaneous evaluation of all possible represented states, have led to the development of approaches that suggest ways to integrate aspects of quantum computing with evolutionary computation (Han and Kim, 2002, Yang et al., 2004). In classical computation, the basic unit of information is a bit, which can be in any one of the two states representing either a one or a zero. In the case of quantum computing the basic unit of information is a qubit or $Q$-bit (Han and Kim, 2002, Yang et al., 2004). A single qubit can not only be in states representing zero or one, denoted $|0\rangle$ and $|1\rangle$ respectively, it can also be in a state which is a linear superposition of the two, i.e., the qubit can be represented as a linear combination of $|0\rangle$ and $|1\rangle$ as $a|0\rangle + b|1\rangle$ where $\alpha$
and $\beta$ are probability amplitudes. In a measurement, the probability of $|0\rangle$ is $\alpha^2$ and that of $|1\rangle$ is $\beta^2$. Thus $\alpha$ and $\beta$ are constrained by the equation $|\alpha|^2 + |\beta|^2 = 1$.

Based on the ideas of qubits and quantum gates, Han and Kim (Han and Kim, 2002) proposed the **Quantum Inspired Evolutionary Algorithm** (QIEA). In QIEA, the basic unit of information is a *Q-bit* which is defined with a pair of numbers $(\alpha, \beta)$ as

\[
\begin{bmatrix}
\alpha \\
\beta
\end{bmatrix}
\]

where $|\alpha|^2 + |\beta|^2 = 1$ and $\alpha^2$ and $\beta^2$ give the probability of the Q-bit being in the zero or one state respectively. A *Q-bit individual* as a string of $m$ Q-bits is defined as

\[
q = \begin{bmatrix}
\alpha_1 & \alpha_2 & \ldots & \alpha_m \\
\beta_1 & \beta_2 & \ldots & \beta_m
\end{bmatrix},
\]

Where $|\alpha_i|^2 + |\beta_i|^2 = 1$, $i = 1$ to $m$. Similar to other evolutionary algorithms, QIEA maintains a population of Q-bit individuals, $Q(t) = \{q_1(t), q_2(t), \ldots, q_n(t)\}$ at generation $t$, where $n$ is the size of population and $q_j^t$ is a Q-bit individual defined as

\[
q_j^t = \begin{bmatrix}
\alpha_{j1}^t & \alpha_{j2}^t & \ldots & \alpha_{jm}^t \\
\beta_{j1}^t & \beta_{j2}^t & \ldots & \beta_{jm}^t
\end{bmatrix},
\]

Where $m$ is the number of Q-bits that make up individual $j$, $j = 1, 2, \ldots, n$. Since we need a binary, or discrete, solution to work with, $Q(t)$ is observed to obtain a population $P(t)$ of binary solutions or strings. A binary solution $x_j^t$, $j = 1, \ldots, n$, which is a string of length $m$, is obtained by selecting a 0 or 1 for each bit using the probability $|\alpha_i|^2$ or $|\beta_i|^2$, $i = 1, 2, \ldots, m$ of $q_j^t$.

In an application of QIEA, binary strings obtained through an observation are taken to encode solutions and are evaluated for fitness.

Combining the ideas in QIEA and the binary particle swarm optimization, Yang et al. (Yang et al., 2004) developed the Quantum (Inspired) Particle Swarm Optimization (QPSO) algorithm. QPSO maintains a population $Q(t)$ of $n$ Q-bit individuals or *particles*:

\[
Q(t) = \{q_1^t(t), q_2^t(t), \ldots, q_n^t(t)\},
\]

Each Q-bit particle $q_j^t(t)$, $j = 1, \ldots, n$, of length $m$ is defined as

\[
q_j^t = \{q_{j1}^t(t), q_{j2}^t(t), \ldots, q_{jm}^t(t)\},
\]
Where $0 \leq q^j_i(t) \leq 1$, $i=1,..,m$. Here $q^j_i(t)$, a Q-bit, unlike the Q-bit in QIEA, represents the probability of the $i$th bit of $j$th particle being zero at generation $t$.

As in the case of the QIEA, we need to carry out an observation to obtain $P(t)=[p^1(t), p^2(t), \ldots, p^n(t)]$, a population of $n$ binary particles, i.e., solutions, from $Q(t)$. This is achieved as follows: for each $q^j_i(t)$, $i=1,..,m$, $j=1,..,n$ generate a random number between 0 and 1. If the number generated is greater than $q^j_i(t)$, then $p^j_i(t)=1$ otherwise $p^j_i(t)=0$, where $p^j_i(t)$ is the discrete bit corresponding to $q^j_i(t)$. In an execution of QPSO, while the termination condition is not achieved, $Q(t)$ is updated to obtain $Q(t+1)$ using the following particle update equations:

$$q^{\text{groupbest}}(t) = \alpha \times p^{\text{groupbest}}(t) + \beta \times (1 - p^{\text{groupbest}}(t))$$

$$q^{\text{selfbest}(j)}(t) = \alpha \times p^{\text{selfbest}(j)}(t) + \beta \times (1 - p^{\text{selfbest}(j)}(t))$$

$$q^{(t+1)} = c_1 \times q^j(t) + c_2 \times q^{\text{selfbest}(j)}(t) + c_3 \times q^{\text{groupbest}}(t) \quad (1.7)$$

Where $\alpha$, $\beta$, $0<\alpha<1$, $0<\beta<1$, are called the control parameters; $c_1$, $c_2$, $c_3$, $c_1+c_2+c_3=1$, $0<c_1<1$, $0<c_2<1$, $0<c_3<1$, represent the degree of belief on the particle in question, the local maximum, and the group (or global) maximum, respectively. $p^{\text{groupbest}}$ and $p^{\text{selfbest}(j)}$ are the best discrete individuals the population and the particle $j$ has met respectively.

The QPSO procedure is described in Figure 1.3 below:

1. $t = 0$;
2. initialize $Q(t)$;
3. observe $Q(t)$ to obtain $P(t)$;
4. evaluate $P(t)$;
5. Update $p^{\text{groupbest}}$ and $p^{\text{selfbest}(j)}$ with the best solution among $P(t)$;
6. while (not termination condition) {
7. \hspace{1cm} $t = t + 1$;
8. \hspace{1cm} update $Q(t)$ using equations in (1.7);
9. \hspace{1cm} observe $Q(t)$ to obtain $P(t)$;
10. \hspace{1cm} evaluate $P(t)$;
11. \hspace{1cm} Update $p^{\text{groupbest}}$ and $p^{\text{selfbest}(j)}$ with the best solution among $P(t)$;
12. }

Figure 1.3. The QPSO Procedure
1.3.4. CLONALG

CLONALG, proposed by Castro and Von Zuben (Castro and Zuben, 2002) is a computational implementation of the clonal selection and affinity maturation principles governing an adaptive immune response. The algorithm takes a systemic view of the immune system and shows that basic immune principles can be used to solve complex tasks. These tasks can be broadly categorized into pattern recognition and optimization tasks. The test data generation problem, as discussed in this thesis, is formulated as an optimization problem, hence we discuss CLONALG in the optimization context only.

Every element of the adaptive, artificial immune system is referred to as an antibody, \( Ab \). Any molecule that can be recognized by the immune system is called an antigen, \( Ag \). On exposure to an \( Ag \), the immune system generates \( Ab \)s whose aim is to neutralize \( Ags \). The initial response to an \( Ag \) is handled by \( Ab \)s present in the immune system which shows different levels of affinity to \( Ag \). Higher affinity antibodies are iteratively cloned and undergo an affinity maturation process to produce \( Ag \) specific \( Ab \)s which then produces an adequate response to the \( Ag \) invasion. Higher affinity \( Ab \)s enters memory pool which then enhances the effectiveness of an immune response to secondary infections. The repertoire of \( Ab \)s is diversified during the affinity maturation process using two mechanisms: hypermutation and receptor editing. When applied to an optimization task, the repertoire of antibodies is not exposed to antigens but an objective function value is computed. Figure 1.4 describes CLONALG for optimization. Each antibody is an encoding of attributes or parameters relevant to the optimization problem. In step 1 a random collection \( Ab \) of antibodies is generated. The objective function value (affinity), also called fitness, for each antibody in \( Ab \) is computed in step 3. A select number of higher affinity antibodies are cloned in proportion to their affinity in steps 5 and 6 to form collection \( C \). Antibodies in \( C \) are subject to hypermutation in step 6 to obtain collection \( C^* \). Hypermutation may be implemented as mutation as in genetic algorithms. The mutation probability is then chosen to be inversely proportional to the affinity. Thus higher affinity antibodies undergo mutation with a much lower probability than lower affinity antibodies. Step 9 implements receptor editing in the form of replacement of \( d \) lowest affinity antibodies. Step 9 is a critical step in introducing diversity in the population and promoting exploration. In the absence of this step, the population may end up converging very quickly to some local optima.

1. Generate a random collection \( Ab \) of \( N \) antibodies;
2. while (termination criterion is NOT met) {

3. Compute affinity of each antibody in \( Ab \);
4. Select \( n, n \leq N \), highest affinity antibodies in \( Ab \) to form collection \( Ab_{[n]} \). Let \( Ab_{(r)} = Ab - Ab_{[n]} \);
5. Clone antibodies in \( Ab_{[n]} \) in proportion to their affinities to form Collection \( C \);
6. Subject antibodies in \( C \) to an hypermutation process inversely proportional to affinity. Let the collection matured antibodies be \( C^* \);
7. Determine the affinity of each antibody in \( C^* \);
8. Replace subset \( Ab_{[n]} \) of antibodies in \( Ab \) with \( n, n \leq N \), highest affinity antibodies in \( C^* \);
9. Replace lowest affinity antibodies in subset \( Ab_{(r)} \) of antibodies in \( Ab \) with new randomly generated antibodies.
10. }

Figure 1.4. CLONALG for optimization

1.4. Organization of the Thesis

This thesis is further organized as follows:

- Chapter II presents an in depth literature review on the application of metaheuristic techniques to the problem of program test data generation and outlines the major issues which this dissertation has addressed.
- Chapter III introduces and presents an experimental evaluation of an improved approach for search-based test data generation with genetic algorithms and quantum inspired particle swarm optimization using branch ordering, memory and elitism.
- Chapter IV describes a modification of a clonal selection algorithm and its experimental evaluation for search-based test data generation for branch coverage.
- Chapter V introduces a combinational strategy for fitness computation for search-based test data generation with genetic algorithm for branch coverage.
- Chapter VI presents and experimentally evaluates an MPI based parallel approach for search-based test data generation with genetic algorithms using the path prefix strategy.
- In order to explore alternate formulations, in Chapter VII a maximization approach is introduces and its experimental comparison with the minimization formulation is presented.
Chapter VIII concludes the thesis with suggestions for future extensions.